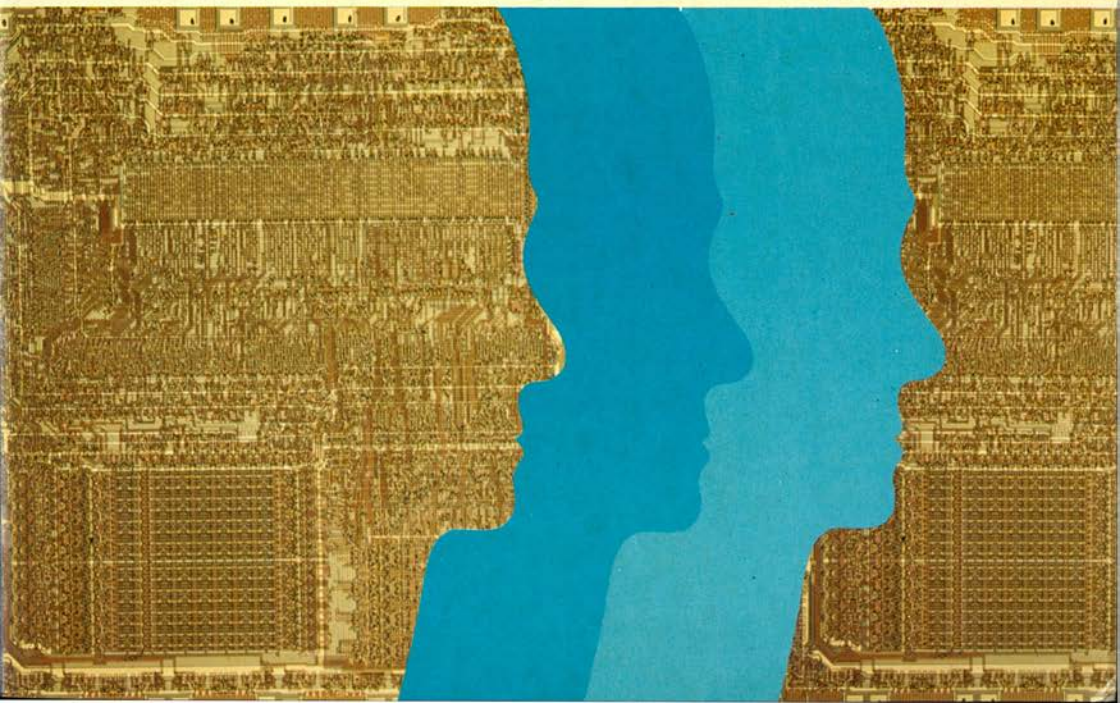
USARE IL MICROPROCESSORE

GIANNI GIACCAGLINI

GRUPPO EDITORIALE JACKSON

USARE IL MICROPROCESSORE

di
Gianni
Giaccaglini

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

PREFAZIONE

Questo testo, che ho avuto il coraggio di scrivere, deve la sua uscita alla disponibilità di molte brave persone giovani, entusiaste e ben disposte che, a tutt'oggi, è dato di incontrare nel mondo effervescente dei micro.
Costoro non solo hanno ascoltato con santa pazienza le mie richieste, e spesso non si sono limitate a sciogliere dubbi e rispondere a quesiti, ma hanno dato, ciascuno per quel che poteva, una mano per risolvere persino problemi di ordine materiale.
L'elenco delle persone cui debbo eterna gratitudine è lungo. Inizia con il tecnico elettronico prof. Boglio, che ha montato, partendo da un kit, il sistemino con il quale ho potuto lavorare. In questa fase è stata pure utile la consulenza (gratuita, o tempora!) dell'ing. Grillo, esperto di microprocessori presso la Homic di Milano. Mi ha poi fatto praticamente da maestro, in tante serate e persino sabati, il Sig. Ellero, tecnico del Centro ricerche di Automatica Enel di Milano: senza di lui non avrei potuto così rapidamente apprendere i piccoli trucchi e "protocolli" della programmazione dello Z-80 e delle sue periferiche. A dattiloscritto ultimato, mi è stata preziosa l'opera di revisione di alcuni capitoli fatta dall'ing. Pranzo Zaccaria, rappresentante della Zelco (concessionaria italiana della Zilog), che ha avuto la bontà di ricevermi sempre con grande cortesia, tra un cliente e l'altro, anche alle ore più tarde, sacrificando qualche week end per leggersi quegli scartofacci.
Due parole soltanto a proposito dell'opera. Essa nasce da un'esperienza professionale e didattica e un fine didattico soprattutto si pone. Sicuramente è anche imperfetta, possiede dei limiti e, probabilmente, qualche errore (la cui segnalazione da parte dei lettori sarà molto gradita). Tuttavia ritengo che qualche vantaggio lo possa presentare: anzitutto tutti i programmini sono stati concepiti in modo originale; poi ho abbondantemente documentato ogni problema sia nell'insieme che nei dettagli (contrariamente ad altre pubblicazioni che spesso forniscono programmi anche strabilianti, ma così scarni nelle spiegazioni da lasciare il sospetto di copiature, oltre ad essere naturalmente poco utili per chi legge). A volte la soluzione proposta non è detto sia la migliore, ma avendo dato quasi sempre un'impostazione aperta e problematica, sono convinto che, almeno i più "provveduti", sapranno escogitarne di migliori, diventando così sempre più bravi, sia attraverso la lettura di altri testi più tecnici ed approfonditi, sia nella concreta attività professionale.
Novara, 1981

L'AUTORE

SOMMARIO

INTRODUZIONE

L'obiettivo che ci poniamo in questo testo è di illustrare, principalmente attraverso esempi di *media* complessità l'uso di un microprocessore. Perciò, non solamente per motivi di spazio editoriale, ma perché ormai esistono nella letteratura anche in lingua italiana un numero cospicuo di opere a carattere introduttivo a livello elementare (v. bibliografia), abbiamo fatto la scelta di dare per conosciute le nozioni fondamentali. Il lettore meno preparato dovrà allora armarsi di pazienza e leggersi un paio di questi testi. La cosa più rapida anzi, che però ci sentiamo di suggerire solo a quelli in possesso di un certo "background" in elettronica e/o informatica, è quella di leggersi il manuale di un microprocessore (nel nostro caso lo Z-80 della Zilog). Questo manuale contiene, sia pure in forma sintetica, tutte le informazioni necessarie (e definite in modo molto esatto e rigoroso). Oltre che essere disponibile presso la Società distributrice dello Z80 è solitamente di corredo al sistema di sviluppo che prima o poi un tecnico o anche un semplice patito dei micro dovrà procurarsi (°) per giocare e ...... soffrire con esso.

L'attività ludica è un aspetto solo apparentemente frivolo di questo settore (si pensi solo ai giochi matematici sviluppati da uomini di scienza estremamente seri). Anzi, una caratteristica ricca di charme del mestiere di programmatore è che ad esso sono richieste doti di inventiva e fantasia creativa, imbrigliate negli stretti binari di una logica ferrea.

L'uso di un sistema di sviluppo è molto importante, non solo a fini di apprendimento. Infatti esso serve, anche nelle applicazioni industriali, per generare i programmi e farne il cosiddetto "debug", cioè un collaudo accurato punto per punto, per trovare e correggere eventuali errori.

Gli esempi che faremo, fanno riferimento ad un sistema del genere (°°).

Essi si dividono in tre tipi:

- programmini di impiego generale (aritmetici, decodifiche, sort ecc.) e altri di puro divertimento;
- programmi di simulazione di controlli (semplici, ma, riteniamo, significativi) di impianti e processi;
- programmi (solitamente di larga massima, più qualche dettaglio significativo) di controlli reali.

La simulazione ha funzione propedeutica per le applicazioni vere e proprie. Si tratta di un discorso rivolto principalmente agli studenti. Ma anche senza poter approfondire l'argomento, vogliamo far notare che anche in campo industriale attraverso la cosiddetta "*in circuit emulation*" si fa, sul sistema di sviluppo, qualcosa di abbastanza simile. Insomma, il gioco diventa serio.

Contiamo infine sul fatto che anche il lettore privo di un sistema di sviluppo identico a quelle cui facciamo riferimento nel testo (o addirittura ... privo di tutto) possa ugualmente comprendere i nostri esempi. Questo anche grazie all'abbondante uso di *diagrammi di flusso* che consentono di passare con facilità *dal problema al sistema*, senza la soluzione di continuità, drammatica per la comprensione, che troppo di frequente si verifica nei progetti scarni nella documentazione.

(°) - Approfittando, come fanno gli hobbisti in questi ultimi tempi, della caduta verticale dei prezzi dei cosiddetti "*personal computers*", che stanno rendendo sempre più democratica l'elettronica.

(°°) - Una scelta del genere era inevitabile, pena il rimanere spesso nel vago. Analoga scelta è stata fatta a favore dello Z80, che a nostro parere, è uno dei più significativi attualmente esistenti, anche perché è molto simile (e più potente) dell'8080 Intel, certamente il micro più diffuso.

BIBLIOGRAFIA ESSENZIALE

MANUALI INTRODUTTIVI

- BUGBOOK V e BUGBOOK VI del Gruppo Editoriale Jackson: in quanto relativi all'8080 Intel, di cui lo Z80 è un parente ... ricco, risultano molto utili con i loro numerosi esempi sia di hardware che di software.

- IL NANOBOOK Z80 Vol. 1 e Vol. 3 pure del Gruppo Editoriale Jackson. Tutti questi manuali fanno esplicito riferimento a sistemi di sviluppò orientati didatticamene (sostanzialmente simili tra di loro ed al tipo descritto da noi).

MANUALI ZILOG

Sono, almeno, necessari:

- Z80 Technical Manual e
- PIO Technical Manual.

Ad essi andranno aggiunti, da parte di chi vuol approfondire il discorso tecnico, altri manuali come quello della SIO, e della CTC ecc.

Si possono ordinare direttamente alla distributrice ZILOG, ossia a:

ZELCO S.r.l. - via Monti, 21 Milano

TESTI APPLICATIVI

- *Z80 Programmazione in linguaggio Assembly*, di L. Leventhal e *Programmazione dello Z80 e progettazione logica* di Adam Osborne, autore di un'altrettanto felice serie di volumi sui microprocessori, edizione italiana Gruppo Editoriale Jackson.

Il testo che citiamo sviluppa nei minimi dettagli e con tutte le varianti possibili un progetto di media complessità (controllo di una stampante seriale) il che offre il destro per insegnare gran parte dei trucchi necessari ad un programmatore di microprocessori (°).

Tecniche d'interfacciamento dei microprocessori di A. Lesea e R. Zaks edizione italiana Gruppo Editoriale Jackson, ottimo per tutti i problemi di hardware. Non citiamo alcun testo di tipo panoramico sui vari tipi di microprocessori, sia perché ormai il loro numero è notevole, sia perché gli argomenti ivi trattati non ci appaiono essenziali per i nostri scopi.

(°) - L'astuto Osborne ha, prima di questo, scritto un testo quasi identico sull'8080: identico il problema, le figure e il 90% delle parole. Solo parte della codifica è stata adattata alla peculiarità dello Z80.

CAPITOLO 1
RICHIAMI DI INFORMATICA
IL FLOW-CHART

In questo capitolo, dopo una serie di richiami ed osservazioni sulla numerazione e sui codici binari, descriveremo quella utilissima ed efficace tecnica volta alla descrizione di procedure e sistemi programmabili che va sotto il nome di *diagrammazione a blocchi* o, più comunemente, di *flow-chart*. Di essa, come già anticipato nella Introduzione, abbiamo intenzione di servirci ampiamente nel seguito.

Sistemi di numerazione e codici binari

Come è ormai quasi universalmente noto, le macchine digitali in genere ed i microprocessori in particolare, sono macchine *numeriche*, che operano su *due* solo stati (fisicamente: "alto" o **H** e "basso" o **L**) detti 0 e 1, ciascuno dei quali costituisce una cifra binaria o BIT.

Sulla base del bit è possibile:

a) rappresentare un numero in base 2 (sistema di conteggio che si può anche chiamare "*binario puro*");
b) raggruppare i bit a n a n, fissando una corrispondenza, detta CODICE binario, tra ognuna delle 2^n configurazioni ed il gruppo di caratteri che si vogliono rappresentare. Quando n = 8, come avviene in molti microprocessori, il raggruppamento di bit si chiama comunemente BYTE.

I numeri binari fino a quindici sono riportati nella seconda colonna della Tavola 1-I. A ciascun bit, come si sa, è associato un valore posizionale basato sulla potenza del due, ossia, dal bit più basso (LSB = Least Significant Bit) al più alto (MSB) : $2^0 = 1$ (unità); $2^1 = 2$ ("duine"); $2^2 = 4$ ("quattrine") e così via, onde un numero binario come 101101, tradotto in decimale, dà: $1.2^5 + 0.2^4 + 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0 = 32 + 8 + 4 + 1 = 45$.

Nella stessa tavola, 3ª colonna, sono riportate le cifre di una numerazione a base sedici, detta perciò ESADECIMALE (inglese Hexadecimal), in cui i caratteri dopo il 9, per convenzione, sono rappresentati con A, B, C, D, E e F, da non confondere con le omologhe lettere alfabetiche.

Dato il perfetto parallelismo tra binario "puro" ed esadecimale, legato al fatto che 16 è una potenza intera del due, esso viene utilizzato per una rappresentazione *compatta* di ogni gruppo di quattro bit, qualunque ne sia il significato (binario puro, codice decimale o di altro tipo, *istruzione* in linguaggio macchina). Poiché il *dato* di un microprocessore come lo Z80 è per l'appunto ad 8 bit, il suo contenuto sarà allora rappresentato da una coppia di cifre esadecimali.

TAV. 1 - I				
N° decimale	**binario "puro"**	**Esadecimale**	**BCD**	**ASCII**
0	0000	0	0000	30
1	0001	1	0001	31
2	0010	2	0010	32
3	0011	3	0011	33
4	0100	4	0100	34
5	0101	5	0101	35
6	0110	6	0110	36
7	0111	7	0111	37
8	1000	8	1000	38
9	1001	9	1001	39
10	1010	A	——	——
11	1011	B		
12	1100	C		
13	1101	D		
14	1110	E		
15	1111	F		

Es. se nell'Accumulatore è contenuto, a partire dal MSB (bit detto A_7) fino al LSB (bit detto A_0) il dato 10110111 in binario, lo potremo compattare come B7:

	MSB							LSB
bit →	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
valore →	1	0	1	1	0	1	1	1
esadec. →	B				7			

L'esadecimale, data la sua compattezza, è anche più rapido per compiere le operazioni aritmetiche binarie. Vediamole rapidamente.

A) Conversione binario-decimale e viceversa

Le potenze del 16 sono: $16^0 = 1$; $16^1 = 16$; $16^2 = 256$; $16^3 = 4096$ (°).

Queste quattro potenze bastano per la conversione di dati contenuti in registri semplici, cioè a 8 bit, o doppi, a 16 bit.

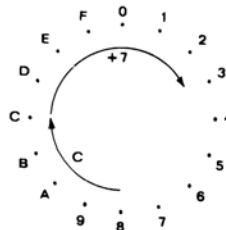
Sia allora l'esadecimale 2FA (ossia, in binario, 001011111010). Otteniamo il decimale: $2 \cdot 256 + 15 \cdot 16 + 10 \cdot 1 = 762$. Identico risultato (provi il lettore diffidente) si ottiene convertendo bit per bit con le potenze del due.

Dal decimale 231, per successive divisioni per 16 con resto si ha invece: $231 : 16 = 14$ con resto 7, onde $231 = 14 \times 16 + 7$ ed i corrispondenti esadecimale e binario sono E7 e 11100111.

È importante far notare che il massimo valore di un byte, *quando si prescinda dal segno* (v. oltre) è $256 - 1 = 255$ (esadecimale FF) e di un doppio

byte $16^4-1 = 64K - 1 = 65535$ (°), valore che corrisponde anche all'indirizzo dell'ultima cella di memoria possibile.

B) Addizione e sottrazione

Servono ad esempio quando si deve calcolare l'indirizzo di una cella di memoria che si trova alcuni posti più avanti o più indietro. Il calcolo in esadecimale ha lo stesso meccanismo che in decimale, si deve solo tener presente che la massima cifra è F e non 9. Es. A + 3 = D (dopo A c'è B, poi C e D); C + 7 = 3 con riporto di 1. Il riporto (ingl. Carry) si ha ogni volta che si passa attraverso lo zero. Il lettore può aiutarsi con la "ruota" esadecimale disegnata accanto. Analogamente, con prestito anziché riporto, si procede con la sottrazione : 32 − 17 = 1B (in esadecimale!) Facendo 7 passi indietro sulla ruota a partire da 2 si arriva a B passando per lo zero, onde, nell'ordine successivo, al 3 si deve togliere 1 e poi ancora 1.

Numeri binari negativi

Si usa la tecnica del complemento a 2. Il bit 7 è quello del segno e vale 0 per numeri positivi, 1 per quelli negativi. In questo secondo caso, gli altri 6 bit costituiscono una rappresentazione *in complemento a due* del numero.

Il complemento a 2 di un numero binario, a n bit, x è il numero che aggiunto a x *dà tutti zeri*. Il complemento a 2 di un byte a 8 bit si può ottenere in vari modi, ad esempio:

1) facendo il complemento a 1 (si invertono tutti i bit) ed aggiungendo 1:

es. 01000110, compl. a 1 = 10111001 + 1 = 10111010

si noti come venga generato automaticamente il bit 1 del segno;

2) togliendo il numero da zero (ignorando il prestito):

```
 00000000 −
 01000110
----------
 10111010
```

Come è noto, con i complementi a 2 sono possibili operazioni algebriche.

(°) - Si ricorda che con l'abbreviazione K si indica il valore $2^{10} = 1024$ onde ad es. 4K corrisponde a 4096 decimale.

Ciò in ultima analisi deriva dal fatto che aggiungendo ad un numero il suo complemento a 2 si ottiene zero o, che è lo stesso, complementando un numero negativo (cioé in complemento a 2) si riottiene il numero positivo.

Nella TAV. 1 - II sono riportati i numeri negativi da − 1 a − 128 in codice esadecimale. I numeri positivi vanno invece da 0 a 127.

Nello Z80 il bit 7 e *copiato* nel flag del segno S, in tutte le operazioni che attivano tale flag. Così, se togliamo 2E da 1E si ottiene F0 (ossia − 16 decimale) e S = 1.

Numeri negativi — TAV. 1-II — ← 2ª cifra hex.

1ª cifra Hex. ↓		F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
	F	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	E	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
	D	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
	C	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	B	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
	A	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
	9	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112
	8	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128

Nota: per ottenere, ad es. − 88, si cerca 88 (per semplicità il segno meno è sottinteso) e si ricava A8.-

Si noti infine che rimanendo nell'ambito dei numeri *senza segno* il campo si estende da 0 a FF, cioé da 0 a 255, in decimale.

Invece della numerazione a base 2, si può desiderare quella decimale. I codici usati sono tanti, ma su 4 bit il più comune è detto BCD (Binary Coded Decimal) o anche codice 8-4-2-1. Esso è riportato nella quarta colonna della TAV. 1-I e, come si vede, coincide con il binario "puro" ma *solo fino a 9*. In BCD, 10, 11 ecc. sono codificati con 8 bit: 0001 0000; 0001 0001 ecc.

In un byte di otto bit generalmente si mettono due cifre decimali. Si dice in tal caso che esse sono *"impaccate"* (packed) nel byte. Per una codifica alfanumerica completa (ossia estesa alle lettere dell'alfabeto ed ai segni speciali) non bastano più 4 bit (°). Un semibyte (o i primi 2 bit se il carattere ne

---

(°) - Ne servono almeno 6, in quanto con 4 bit si ha un totale di $2^4 = 16$ caratteri codificabili, mentre con 6 si passa a $2^6 = 64$. Con 8 bit si ha la bellezza di 256 combinazioni.

ha 6) è allora adibito a *zonatura*. Nei grossi elaborati (es. IBM 370) la zonatura di una cifra è l'esadecimale F (questo codice è detto EBCDIC).

Nei micro, è abbastanza frequente, soprattutto nelle interfacce con TTY (TeleTYpe) o CRT display (visualizzatori Cathode Ray Tube, monitor o simili) il codice ASCII, in cui la zonatura delle cifre è 3 (v. col. 5 TAV. 1-I).

Altri caratteri ASCII di impiego corrente sono riportati nella TAV. 1-III

TAV. 1-III

Caratteri ASCII più comuni (codifica esadecimale)

Caratt.	Codice	Caratt.	Codice	Caratt.	Codice
A	41	N	4E	+	2B
B	42	O	4F	−	2D
C	43	P	50	*	2A
D	44	Q	51	/	2F
E	45	R	52	.	2E
F	46	S	53	,	2C
G	47	T	54	;	3B
H	48	U	55	:	3A
I	49	V	56	!	21
J	4A	W	57	?	3F
K	4B	X	58	=	3D
L	4C	Y	59	(	28
M	4D	Z	5A	)	29

Ad essi va aggiunto il *blank* (spazio vuoto) rappresentato a volte nelle minutazioni manuali dei programmi con il simbolo ƀ (b minuscola con taglio sul gambo). Esso ha codifica 20.

A questo punto, è evidente che occorre una convenzione per evitare confusioni (o di dover ogni volta precisare il sistema di numerazione o codice, come fatto fin qui).

La notazione più pratica è quella dell'Assembler:

- *numeri decimali:* una D sulla destra o anche nessun segnale: es. 1397D o solo 1397;
- *numeri esadecimali:* si aggiunge H sulla destra; es. 3AH;
- *codici* ASCII: si scrivono i caratteri *in chiaro*, inseriti tra apici: es. 'TUTTO OK' equivale a 545554544F204F4BH si noti il 20H del blank).
- *numeri binari:* si aggiunge B a destra; es. 10011100B, equivale a 9CH

Automi, algoritmi, diagrammi di flusso (flow-chart)

Nella teoria dei sistemi, si parla di AUTOMI, che, senza pretesa di una definizione rigorosa, possiamo denominare come ogni entità (macchina in particolare) di cui si possano descrivere gli stati ed il loro successivo evolvere.

Quando questo sistema è descrittibile mediante un ALGORITMO, ossia un procedimento fatto di passi successivi tutti legati logicamente tra di loro, la teoria mostra l'equivalenza tra automa e algoritmo. Poiché, come stiamo per vedere, algoritmo e flow-chart, o meglio programma, finiscono per essere sinonimi, ecco spiegato, in modo perlomeno intuitivo come risulti in linea di principio sempre possibile realizzare per mezzo di una macchina programmabile - cioé con del *software* supportato da un elaboratore - quanto invece tradizionalmente trova implementazione mediante dispositivi hardware (o, come si dice con riferimento a congegni digitali elettronici, dispositivi a *logica cablata*).

D'altronde il flow-chart trova già da tempo applicazione non solo nella descrizione ma anche in concrete procedure di analisi e sintesi di circuiti logici sequenziali sincroni (ed anche asincroni) quali quelli formati da ROM e flip flop D-type o J-K.

I simboli fondamentali del flow-charting sono in figura 1-1.

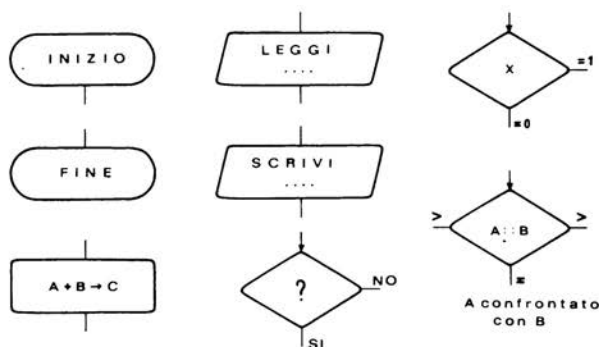



Figura 1-1 Simboli dei diagrammi a blocchi.

L'ovale schiacciato, corredato da scritte opportune dentro di esso, come START,STOP, INIZIO, FINE ecc. segna i *limiti* del diagramma.
Il blocco rettangolare serve a rappresentare operazioni di elaborazione interna (da parte della unità centrale o CPU, nel caso di un elaboratore).

Il parallelogramma evidenzia operazioni I/O (Input/output), mentre il rombo costituisce il simbolo di *decisione*. Esso ha una entrata e fino a tre diramazioni di uscita (nel caso di un confronto, rappresentato di solito con 4 puntini posti tra i termini paragonati, si possono dovere distinguere i casi $>$; $<$ e $=$).

Più comunemente, le uscite sono due: $=$ contro $\neq$; $>$ rispetto a $\leqslant$; $<$ verso $\geqslant$.

In diagrammi di tipo generico, si preferisce scrivere entro questo blocco una domanda, con le due uscite che portano le risposte SI o NO. Nel caso poi di una variabile che può assumere solo due valori (esempio tipico 0 o 1) si usa spesso segnare i due valori su ciascuna delle due uscite: v. la variabile X della Fig. 1-1.

Sulla scorta di questi elementi, ai quali va aggiunta la *linea di collegamento*, orientata con frecce quando si debbano evitare confusioni, che unisce blocchi rappresentanti operazioni consecutive, vediamo ora un primo esempio che illustra un algoritmo visto come puro modo di operare, senza riferimento alcuno ad un elaboratore (v. Fig. 1-2).

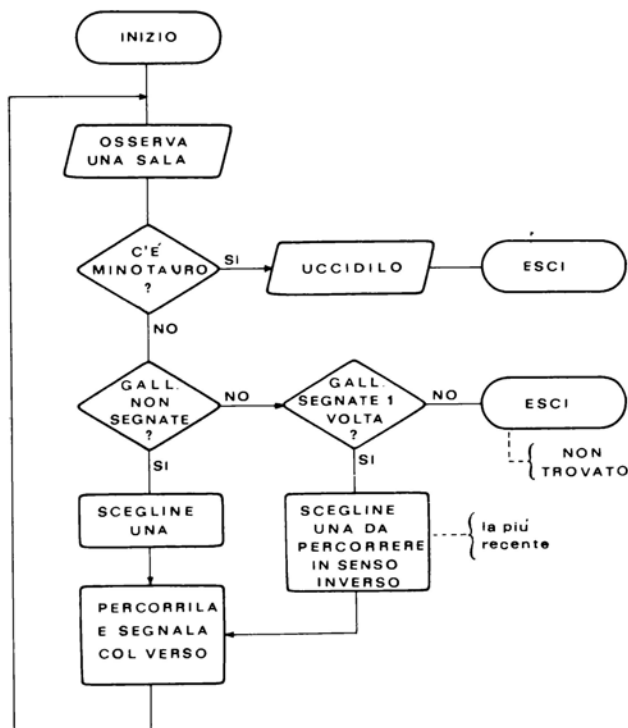



Figura 1-2 Teseo alla ricerca del Minotauro.

Si tratta di *Teseo alla ricerca del Minotauro*. Il labirinto è un intrico di sale connesse da gallerie. Teseo qui si serve di un gessetto e, ad ogni sala, sceglie una galleria e la percorre segnandola con il verso di percorrenza e con un numero progressivo. Nella scelta suddetta, dà la precedenza a corridoi non segnati ancora e, qualora non ne trovi alcuno non segnato ne sceglie uno *da percorrere in senso inverso e segnato col numero più alto* tra quelli già

percorsi una volta sola (la scelta più semplice, dove possibile, cioé quando non sia già stato percorso due volte, è di tornare indietro nel corridoio di arrivo). Si comprende (ed il lettore potrà convincersene disegnandosi un labirinto con o senza Minotauro e mettendosi nei panni dell'eroe), che alla fine Teseo o raggiunge il Minotauro oppure, arrivando ad una sala con tutte le gallerie segnate due volte, ha la certezza che il mostro è irraggiungibile (magari perché è solo un'invenzione propagandistica).

Si possono dare flow-chart di questo tipo, anche scherzosi (°). Più utili ai nostri fini risultano quelli volti alla descrizione di macchine e impianti. Ne mostriamo, sempre in forma generica, uno semplicissimo nella Fig. 1-3, relativo ad un controllo di temperatura.

Si prevede la scansione ciclica (o "*polling*") del sensore che rileva la grandezza che il regolatore deve mantenere costante.

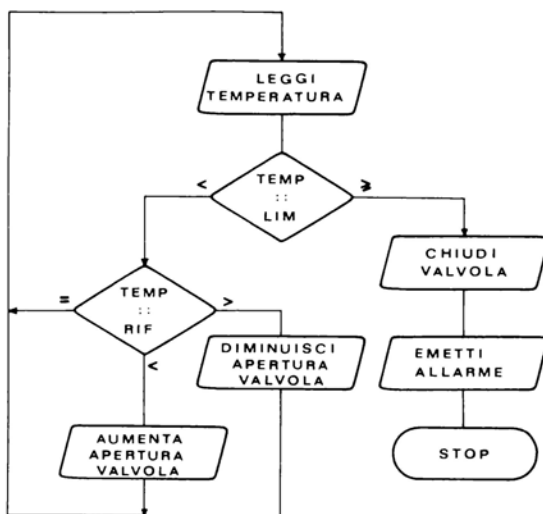



Figura 1-3 Controllo di temperatura (di larga massima).

Si opera un primo confronto con un valore di sicurezza LIM e, se esso è superato, si chiude la valvola del bruciatore e si lancia un allarme. Si confronta poi col *riferimento*, ossia il valore conforme al corretto funzionamento: dal diagramma a blocchi è immediato comprendere il tipo di azione che viene previsto nel caso che, rispettivamente, il valore rilevato supera, è inferiore o eguaglia il riferimento.

(°) - Es. flow-chart del risveglio dell'uomo sposato, che prevede il bacio alla consorte se e solo se il matrimonio data da meno di due anni (la fonte - ci credereste? - è la Honeywell!).

Questo diagramma è molto generico e semplificato. In particolare manca la parte relativa alla messa a punto iniziale dell'impianto. Anche così, comunque, esso si presta per un primo approccio al problema, facilitandone la comprensione. Si noti l'uso di simboli di I/O in riferimento ad operazioni di ricezione/trasmissione di informazioni o comandi rispetto al mondo esterno al sistema di controllo (°).

Con gli esempi di simulazione e, poi, con quelli di realizzazione effettiva, contiamo in seguito di rendere più concreti possibile i discorsi sui controlli mediante microprocessori.

Vediamo ora invece alcuni esempi semplici relativi ad elaborazione dati. Infatti un micro può anche essere impiegato come calcolatore programmabile vero e proprio, ma anche nelle applicazioni di controllo può capitare -

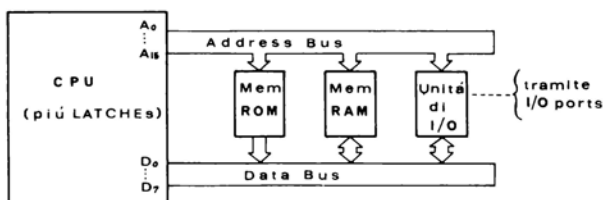



Figura 1-4 Schema a blocchi generale di un sistema a micro-P.

almeno nei sistemi più sofisticati - di aver bisogno di elaborazioni di tipo calcolistico anche molto complesse. (°°).

Per fissare le idee, facciamo riferimento (Fig. 1-4) alla schematica struttura di un sistema a microprocessore (il lettore farà anche meglio se si documenterà più a fondo su questi problemi nei manuali introduttivi che abbiamo consigliato).

Come è noto, la CPU, tramite l'*Address bus* (linea $A_0 \div A_{15}$) può indirizzare, cioé selezionare, posizioni di memoria o dispositivi di I/O (PIO, SIO ecc., nel caso di sistemi Zilog) per l'interfaccia con l'esterno.

Il primo caso si ha con le istruzioni di elaborazione interna (trasferimento, aritmetiche ecc.) che interessano, per l'esattezza, anche i registri di cui dispone la CPU. Il secondo caso si presenta con le istruzioni di Input e/o Output (in Assembly Z80: IN e OUT).

(°) - Anche in Fig. 1-2 UCCIDILO è stato messo entro parallelogramma, anche se lì, a dir il vero, queste sottili distinzioni non hanno molto senso.

(°°) - Es. filtri digitali, FET (Fast Fouriere Transform), calcoli statistici sui segnali ecc., tutte applicazioni queste che ovviamente esulano dai nostri fini, anche perché sono numerosissime quelle in cui i calcoli numerici sono ridotti al minimo.

I dati, esistenti sul *DATA BUS* (linee $D_0 \div D_7$) interessano, nei due casi la memoria o l'I/O. Con le memorie di tipo ROM (EPROM, nel caso più frequente) si può fare solo la lettura, ossia dalla EPROM, tramite il Data Bus, all'Accumulatore della CPU. Con le RAM invece si può anche scrivere nella cella indirizzata dall'Address Bus.

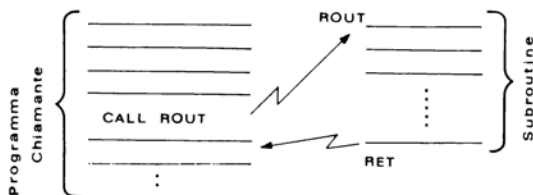
Si ricorda l'uso di ROM e di RAM: le prime contengono il cosiddetto FIRMWARE, ossia solo istruzioni fisicamente inalterabili (tranne che con mezzi fisici opportuni, estranei comunque alle possibilità della CPU). Le RAM possono invece contenere sia istruzioni che *dati-da-elaborare*.

Bisogna allora distinguere due possibili applicazioni:

- micro calcolatori programmabili: il firmware costituisce il cosiddetto MONITOR, base del sistema operativo o di gestione, mentre nelle RAM si scrivono, oltre ai dati, programmi di volta in volta diversi;

- controlli a μP: il firmware di norma costituisce tutto il programma e la RAM, a volte di portata limitata, serve solo per i dati (la cosiddetta area SCRATCHPAD), nonché per realizzare lo *stack*.

Un caso intermedio è quello dei SISTEMI DI SVILUPPO, in cui prevale la RAM, come nei microcomputer (°): i programmi destinati a sistemi di controllo sono prima scritti nelle RAM del sistema di sviluppo, sottoposti a "debugging" e, una volta collaudati, trascritti su EPROM.

Con riferimento allora al generico sistema della Fig. 1-4, andiamo ad illustrare alcune semplici routine di calcolo.

Si useranno solo simboli rettangolari, in quanto i dati si suppongono già nella memoria o nei registri della CPU. Negli ovali di terminazione scriviamo, invece, ENTRY all'inizio e RET nella (o nelle) uscite.

La RET è infatti l'istruzione terminale di una subroutine che, come è notorio, provoca il rientro automatico al programma *chiamante*, come illustrato qui a fianco.

(°) - E, quando la RAM ha dimensioni ragguardevoli, possono anche servire da microcalcolatori (*personal computer* ad esempio). Si comprende allora che in questi casi la distinzione viene più a dipendere dall'uso che si fa del sistema che dalla macchina in sé.

Ogni riga sta a rappresentare una generica istruzione. Quando, nel programma chiamante (ossia nel gruppo di istruzioni in cui si ha la richiesta della subroutine) si incontra una istruzione come la CALL ROUT, essa forza nel PC (Program Counter) l'indirizzo della prima istruzione di ROUT.

Il valore della istruzione *successiva* a CALL RET viene caricato sullo stack. Si va quindi ad eseguire ROUT, al termine la RET "scarica" dallo stack quell'indirizzo ed il programma chiamante riprende con quella successiva istruzione.

Dentro un blocco rettangolare scriveremo istruzioni del tipo seguente, tenendo conto che con A, B o anche PIPPO, PIO, DATO ecc. indicheremo dati contenuti in registri o aree di memoria, mentre la freccia è orientata verso il dato che costituisce il risultato:

		A	→	B	(trasferimento di A in B)
A	+	B	→	C	(somma di A + B con risultato in C)
A	+	1	→	A	(incrementa di 1 A)
A	+	N	→	A	(incrementa A di N)
B	−	2	→	B	(decrementa di 2 B) ecc.

Primo esempio: *moltiplicazione di due interi tramite somme ripetute*

Un dato MDO (MoltiplicanDO) è da moltiplicare per MRE (MoltiplicatoRE). Il risultato va nel dato PR (PRodotto). E' necessaria una routine in quanto nei microprocessori a 8 bit non esiste una istruzione singola di moltiplicazione, né tantomeno di divisione.

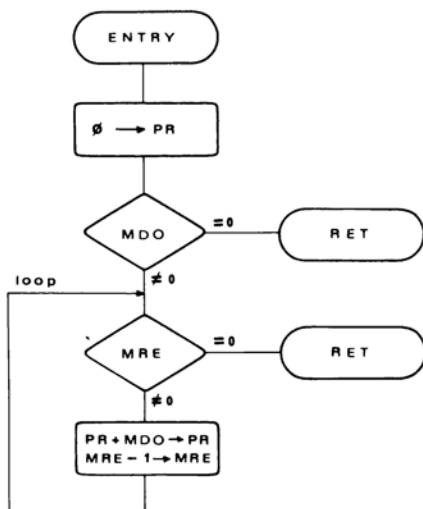



Figura 1-5 Prodotto di interi per somme ripetute.

L'algoritmo più semplice (un altro migliore e, soprattutto, più veloce lo vedremo in seguito) consiste nell'aggiungere al dato PR, *fatto inizialmente uguale a zero*, il dato MDO tante volte quante ne indica MRE. Questo, perciò, è ogni volta decrementato, fino a quando, con MRE = 0, il ciclo ha termine.

Per prima cosa, dopo 0 → PR, si interroga MDO: se esso è zero, si evita di aggiungere MRE volte zero, uscendo con il valore PR = 0. Lo stesso si fa con MRE. Questo confronto viene però inserito nel "*loop*" (= ciclo), in cui, oltre ad aggiungere MDO a PR, si toglie 1 a MRE e ci si riallaccia a monte del confronto di MRE con zero: quando MRE = 0 si esce dalla routine.

Secondo esempio: *divisione* (intera tra interi) *per sottrazioni successive*

Qui DDO è il DividenDO, DRE è il DivisoRE.

La prima parte serve ad evitare la divisione per zero. Se DRE = 0, si esce con SW = 1. SW (sta per SWitch) è spesso un singolo bit o anche un flag del

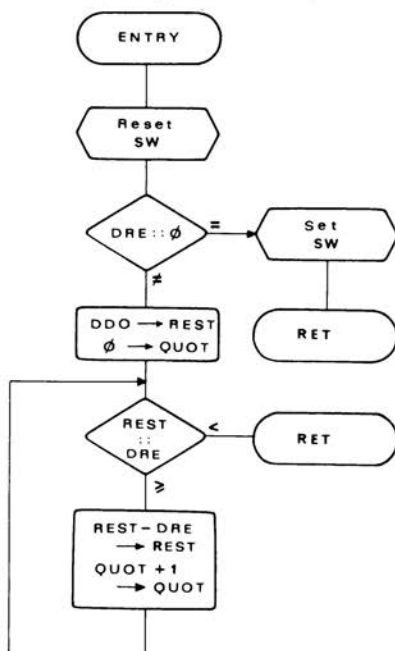



Figura 1-6 Quoziente (e resto) tra due interi per differenze ripetute.

registro F (es. il riporto *Carry*). Porre un flag a 0 o a 1 si indica allora con "*reset*" e, rispettivamente, con "*set*". Nel programma chiamante al rientro dalla routine basterà allora vedere se SW = 0 oppure 1, per stabilire il da farsi (es. arrestare l'elaborazione per segnalare errore).

Il dato DDO è assunto come resto REST, inoltre si pone zero nel campo QUOT (campo è il nome generico che si dà ad un'area di memoria o registro contenente un dato). Se REST è minore di DRE, si esce dalla routine, in questo caso con REST = DDO e QUOT = 0 (come è giusto: es. 12 : 15 = 0 con resto 12). In caso contrario, si toglie DDO al REST e si aggiunge 1 a QUOT (il divisore DRE "ci sta" almeno una volta) e così via, finché REST $<$ DRE (es. 17 : 3 dà luogo a 5 successive sottrazioni di 3, finché REST = 17 $- 3 \cdot 5 = 2$ e QUOT = 3).

Questi primi esempi, pur nella loro estrema semplicità, contengono già concetti essenziali come quello di LOOP (e di procedimento *interativo*), contatore, decontatore, totalizzatore (un campo come PR, fatto inizialmente uguale a zero ed incrementato ripetutamente di un altro dato).

Nelle applicazioni di controllo può anche capitare di aver a che fare con cicli *chiusi su se stessi*, dai quali cioé, di norma, non si esce mai. Ad esempio, in figura 1-3, se non si supera la temperatura LIM, si continua a ciclare col polling della temperatura ecc., indefinitamente.

Da questo semplice esempio si può allora già comprendere che gli INTERRUPT, come il dio di Voltaire, se non esistessero, bisognerebbe inventarli. Un interrupt, come è noto, è un segnale fisico esterno che interrompe, appunto, l'esecuzione del programma in corso (salvando al solito sullo stack l'indirizzo dell'istruzione seguente), per passare ad altre funzioni. Ciò è indispensabile quando l'intervento che si richiede deve essere più rapido possibile. Sempre con riferimento alla figura 1-3, si potrebbe pensare di sostituire il confronto TEMP con LIM, affidando ad un dispositivo hardware questa funzione (es. un congegno a soglia opportunamente tarato). Quando questo dispositivo scatta, viene generato un segnale di interrupt ed il controllo salta alla routine che provvede a lanciare l'allarme e ad altre operazioni consimili.

Analogamente, per passare alla funzione (essenziale, anche se trascurata per semplicità nella figura in parola) di arresto impianto a fine esercizio, si può pensare ad un altro interrupt, se non altro per evitare la noia di un ulteriore "polling" dell'ingresso da cui proviene tale comando (analogo al polling della TEMP).

Terzo esempio: *trovare la media* (MED) *di dati provenienti da una periferica*

Dopo aver inizialmente azzerato i campi N e MED, si esegue LEGGI DATO che sta a indicare una opportuna routine che provvede a caricare, dalla periferica, la nuova quantità (°) in DATO (nel caso più comune si trat-

---

(°) - Per periferica si intende un dispositivo come un lettore di scheda a simili o, più banalmente, una tastiera.

ta dell'Accumulatore). Questa è cumulata in TOT e, insieme, si incrementa il contatore N. Quando arriva il codice FF (esadecimale) che si suppone segnale di *fine dati* si calcola la media dividendo TOT per N.

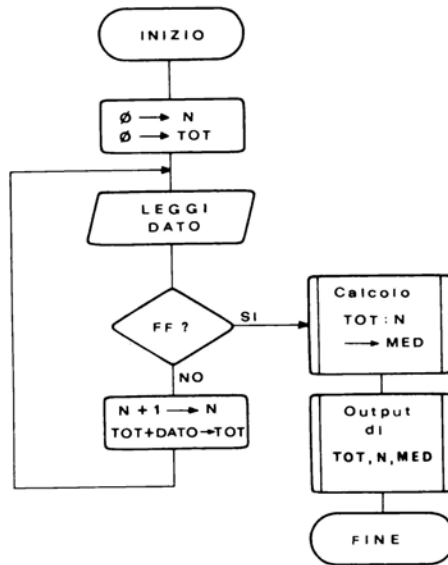



Figura 1-7 Media dei dati provenienti da periferica.

Si approfitta per far notare il simbolo di *sottoprogramma* (rettangolo con barre laterali). Esso non coincide propriamente con il concetto di subroutine in senso stretto (per il quale, comunque, preferiamo esplicitamente indicare CALL + nome subroutine). Semplicemente esso serve, per comodità, quando non si vuole appesantire il diagramma d'assieme. I dettagli del sottoprogramma sono evidenziati a parte o su altro foglio.

Vogliamo ora introdurre l'importantissimo concetto di INSIEME di dati (o *vettore*), detto più volgarmente TABELLA (°). Si sfrutta l'indirizzamento *indiretto* tramite registri *puntatori* (le coppie HL, DE e BC nello Z80 e nell'8080) o *registri indice* (IX e IY, nello Z80).

In tal modo, quando un puntatore o indice - chiamiamolo genericamente k - ha il valore corrispondente ad un certo *indirizzo* di memoria, è il *contenuto* della cella di quel tal indirizzo ad essere chiamato in gioco.

Il puntatore o indice viene messo tra parentesi: con EL (k); TAB (k) ecc. si indicherà quindi il contenuto della cella puntata da K.

(°) - Ci si limita qui agli insiemi ad una sola dimensione. Per quelli a 2 o più indici (matrici) si rimanda a testi più approfonditi.

Questa tecnica è essenziale nell'elaborazione dati. Essa consente, manipolando gli indici (ad es. incrementandoli o decrementandoli) di utilizzare ripetutamente - in loop - la stessa istruzione.

Quarto esempio: *trovare il massimo* (MAX) *e il minimo* (MIN) *valore di un insieme*

Il problema strutturalmente rassomiglia a quello dell'esempio precedente (e il lettore potrà fare un'ibridazione tra i due: massimo, minimo e media di un insieme, oppure in una serie di dati introdotti dall'esterno) solo che al posto di un'operazione di input c'è ora l'incremento di un puntatore k ed il suo posizionamento iniziale al valore - qui genericamente detto INIZ - pari

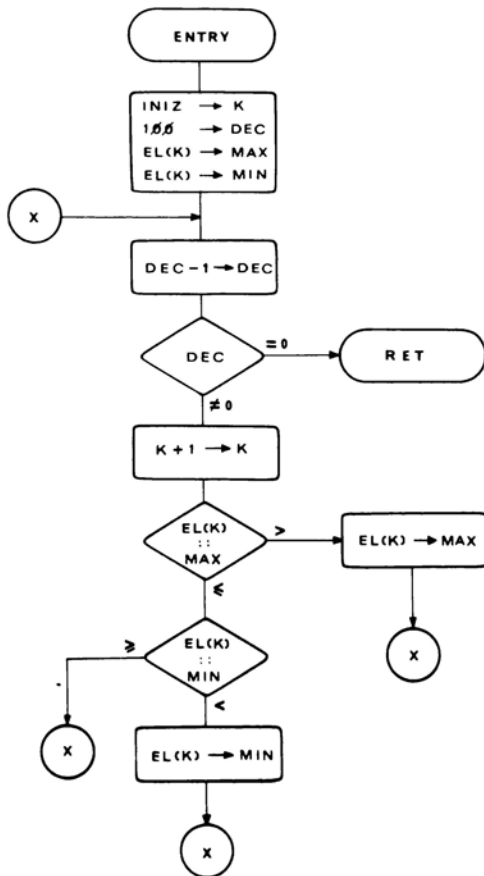



Figura 1-8 Trovare il MAX e il MIN in un insieme.

all'indirizzo della prima cella. Bisogna anche conoscere la *dimensione* dell'insieme. Sia essa, per fissare le idee, di cento elementi.

Dopo aver messo INIZ in k e 100 in DEC, si parte caricando EL(k) sia in MAX che in MIN, cioé assumendo il primo elemento come massimo e minimo. Ciò non è una stranezza, anzi è logico, in quanto se la tabella avesse un solo elemento esso ne sarebbe il massimo e il minimo.

Si decrementa quindi DEC e, se diverso da zero, si passa all'elemento successivo, incrementando k. Ora l'elemento EL(k) è il 2°, poi il 3° e via dicendo. Esso viene confrontato prima con MAX,poi con MIN, affinché, se esso merita il posto di uno dei due, lo prenda (istruzioni di trasferimento di EL(k) in MAX e in MIN). Quando, dopo 100 giri DEC = 0 la routine ha termine.

Si noti come il posizionamento di DEC − 1 e il successivo confronto di DEC con zero a monte dell'incremento di k è corretto: infatti con una tabella di un solo elemento si avrebbe valore iniziale di DEC = 1 e, come si è visto, l'unico EL sarebbe il MAX e il MIN. Il lettore, comunque, si abitui, quando ha dubbi, a eseguire passo passo un flow. Un tale debug "in bianco" per quanto noioso e pedestre, risparmia spesso amare sorprese nel seguito.

In questo diagrammino si è fatto uso del simbolo *connettore*, cerchietto con su un simbolo che richiama un circoletto recante lo stesso simbolo. Serve a sostituire la linea di collegamento ed è indispensabile quando i simboli da collegare sono troppo lontani o addirittura su fogli diversi, o se si vogliono evitare intrecci di linee.

In fig. 1-8 si hanno tre connettori X "in partenza" (freccia *verso* il circoletto) che fanno capo ad un unico X "in arrivo" (in alto, a monte di DEC -1). Quest'ultimo deve sempre essere unico. Il connettore può in certo senso dare un'idea visiva della fondamentale istruzione di salto.

Quinto esempio: *ordinamento* (inglese SORT) *di una tabella*

Si vogliono disporre in ordine crescente gli N elementi di una tabella, supposti inizialmente messi a caso.

L'algoritmo che si propone, uno tra i tanti possibili, fa uso di due puntatori j e k, il secondo sempre di una unità maggiore dell'altro. Posto all'inizio il primo, j, al valore IND (cioé INDirizzo prima voce) ed il secondo al valore IND + 1 dell'indirizzo della voce accanto, nonché assegnato al decontatore B il valore N-1 (tante, infatti, sono le *coppie* di elementi da confrontare), si paragona una coppia di elementi contigui mediante l'istruzione EL (j) comparto con EL (k). Se i due sono a posto (il primo minore o uguale di quello accanto) si va avanti decrementando B ed incrementando j e k poi ripetendo il confronto per la successiva coppia cui così si viene a puntare. Se ad un cer-

to punto si trova una coppia *fuori posto* si provvede alla *scambio* dei due elementi. A tale scopo è necessaria un'area o registro di deposito DEP la cui funzione è illustrata qui sotto.

EL(j)	EL(k)	DEP	
x	y	–	dopo EL(j)→DEP si ha:
x	y	x	dopo EL(k)→EM(j) risulta:
y	y	x	infine, dopo DEP→EL(k):
y	x	x	

A questo punto, la cosa più semplice è ricominciare da capo, in quanto lo scambio ha portato in avanti un elemento che potrebbe aspirare a un ulteriore....avanzamento meritocratico. Operando in questo modo, quando si giunge a B = 0 si è sicuri che: a)nessuno scambio è stato necessario; b)ogni elemento è maggiore del precedente e quindi la tabella è sistemata.

Per illustrare più da vicino il meccanismo, supponiamo una tabellina di cinque elementi di valore, nell'ordine 6,7,2,4,5. Per semplicità, facciamo IND = 1. Si comincia con j = 1 e k = 2, mentre B = 5 - 1 =4. J e k puntano agli elementi di valore 6 e 7, che sono a posto. Si passa allora a D = 3, quindi a j = 2 e k = 3. I corrispondenti elementi, che valgono ora 7 e 2 vanno scambiati e la tabellina diventa: 6,2,7,4,5. Si ricomincia con j =1; k =2; B =4. Ora la prima coppia non è a posto e si fa uno scambio che dà 2,6,7,4,5. Ricominciando da capo e andando avanti si arriva con B = 2; j = 3; k = 4 allo scambio di 7 con 4 ecc. ecc.
All'ultima spazzolata la tabella è ormai in ordine: 2,4,5,6,7. Si inizia al solito con B =4; j = 1; k =2. Tutte le coppie sono trovate in ordine, quindi ci basterà vedere cosa accade man mano che B diminuisce. La cosa è illustrata nella tabellina che segue.

Reg. B	situaz.
4	j=1 k=2
3	j=2 k=3
2	j=3 k=4
1	j=4 k=5
0	RET

L'uscita, con B = 0, dalla routine si colloca al punto giusto per evitare un confronto (privo di senso e pericoloso) tra l'elemento quinto e il successivo (che *non* fa ovviamente parte della nostra tabella!)

Abbiamo insistito un po' con questo "debug sulla carta" perché il lettore si rendesse meglio conto di cosa intendevamo dire quando poc'anzi ne abbiamo parlato.

Terminiamo illustrando un espediente di programmazione che va sotto il nome di SWITCH o *deviatore*. La sua manipolazione, solitamente il set o il reset o anche, come pure si usa dire la sua messa in ON o OFF (tutti sinonimi dei valori 1 e 0) è evidenziata nei flow-chart con un simbolo esagonale schiacciato.

La messa in ON di uno switch ha in generale lo scopo di memorizzare una certa situazione, in modo che, più avanti nel programma, l'interroga-

zione dello stato del deviatore serva come alternativa per seguire una strada anziché un'altra (v. Fig. 1-10 a) (°).

La funzione del deviatore è sostanzialmente quella di compattare un programma. Un possibile uso è nella stessa figura, in b. Una serie di istruzio-

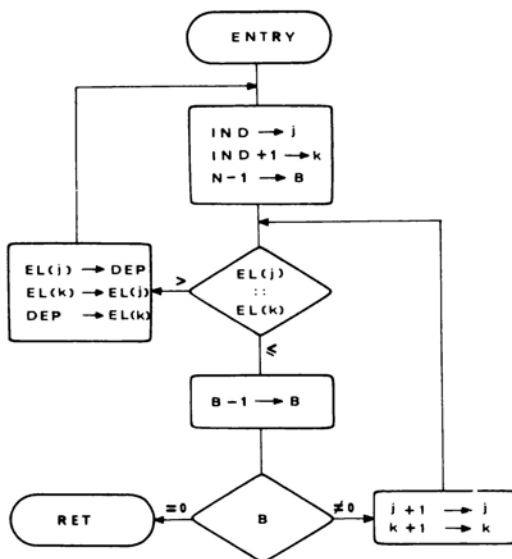



Figura 1-9 Orientamento (sort) di un insieme.

ni X è seguita da un'altra seria Y, poi si ha ancora X. Si supponga che il numero di istruzioni del sottoprogramma X sia abbastanza alto da far ritenere utile il poterle scrivere una sola volta. Introducendo lo switch SW, come indicato in c, esso è messo a zero *prima* della esecuzione di X, subito dopo è interrogato. La prima volta che è eseguito X, SW = 0 ed il programma è deviato sulla destra. Qui si mette in ON il deviatore quindi si esegue Y poi si salta, per una ri-esecuzione, a X, al termine del quale, però, stavolta SW devia al punto 2 (che è l'omologo della versione elementare disegnata in b).

La maggior compattezza di c rispetto a b si paga con una maggior complicazione di interconnessioni. A questa soluzione, poi, si preferisce la tecnica della chiamata di subroutine, che ha il vantaggio di una maggiore chiarezza (°°).

(°) - Bisogna naturalmente stare attenti agli esatti momenti in cui un deviatoio va acceso o spento. La pratica e la logica suggeriscono caso per caso come operare.

(°°) - Detto senza offesa per chi non avesse ancora capito, le subroutine servono principalmente per risparmiare istruzioni. Es. la routine della moltiplicazione, *scritta una volta sola* può essere chiamata da più punti del main program (progr. principale), cioé tutte le volte che occorre tale operazione

I deviatori possono invece essere utili *associati ad una subroutine.*

Un primo esempio, già visto in Fig. 1-6 si ha quando, in una subroutine si hanno uscite *multiple*: lo switch può allora servire a dare al programma chiamante la nozione di quale sia l'uscita di provenienza o, in altri termini,

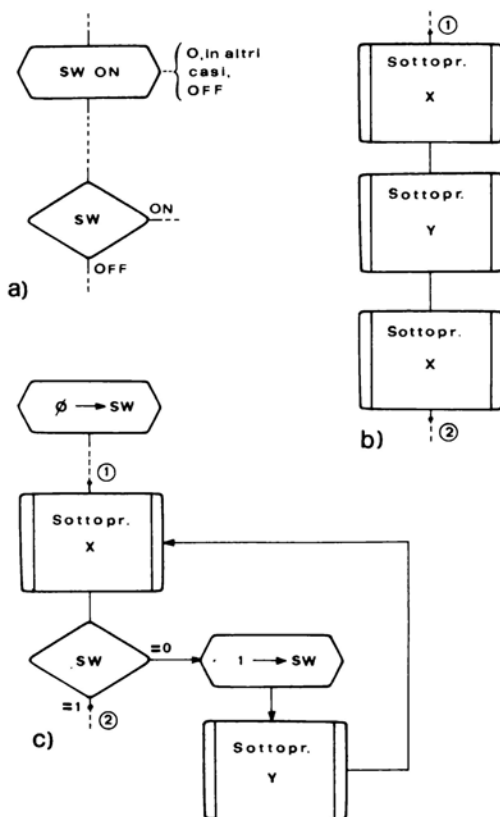



Figura 1-10 Uso del deviatore.

quale particolare situazione si è verificata (si riveda la Fig. 1-6 in cui SW = 1 segnala divisione per zero).

Un altro caso, opposto, consiste nel posizionare il deviatore nell'ambito del programma chiamante, per far sì che la subroutine possa nel suo interno operare, a seconda dei casi, in modi diversi. Si risparmiano istruzioni in quanto si evita di scrivere due distinte subroutine, magari pochissimo diverse tra di loro.

Molte sarebbero le cose e gli esempi ancora da dire e illustrare.

Contiamo tuttavia di ampliare e dettagliare a sufficienza il discorso in tutti i capitoli che seguono.

"L'uomo non dispone di altro per esprimersi: o la prosa o i versi" (Molière, Il borghese gentiluomo, atto II)

CAPITOLO 2

IL LINGUAGGIO ASSEMBLY Z80

Un microprocessore può essere programmato direttamente in linguaggio macchina, con i codici operativi scritti in esadecimale (o, in altri casi, in ottale o persino in binario!). Ciò, anche se scomodo, può essere didatticamente utile, per conseguire una conoscenza molto accurata della macchina. Tuttavia, una volta scritto in questo modo, il programma risulterebbe comunque pressoché illegibile anche alla medesima persona che lo ha scritto.

I codici "mnemonici" (o, come anche si dice, simbolici) *Assembly* sono allora senz'altro indispensabili per sapere il significato delle varie istruzioni in modo più chiaro e diretto. Pertanto scriveremo tutti i nostri programmini in Assembly language, riportando alla sinistra di ogni istruzione in simbolico i codici assoluti corrispondenti. Es.:

```
21 10 0C        LD HL, 0C10H
```

dove LD HL, 0C10H significa carica (LD abbreviazione di LoaD) nella coppia di registri HL l'esadecimale 0C10, con le graffe e le frecce che stanno ad indicare codici assoluti e mnemonici corrispondenti.

Come dovrebbe essere noto, un linguaggio simbolico è un insieme di regole per scrivere un programma nel modo più chiaro per l'utilizzatore.

Si usano termini alfanumerici anziché numerici, sia per i codici operativi (es. ADD per la somma) che per gli *operandi*. Quest'ultimo fatto è particolarmente vantaggioso, in quanto svincola dalla necessità di effettuare noiosi calcoli, sia per quanto riguarda i dati in memoria, sia, soprattutto, per quanto concerne i *salti* di programma. Ai primi e ai secondi, in Assembly Z80, è lecito assegnare nomi simbolici detti LABEL (etichette).

Esempio 1. Anziché dire LD A, (0C50H) (cioè carica nell'Accumulatore A il contenuto della cella di memoria RAM di indirizzo 0C50H si potrà an che dire: LD A, (PIPPO), ove PIPPO è la label associata da noi all'indirizzo di quella cella e che naturalmente l'assemblatore, in fase di traduzione, converte in 0C50H.

Esempio 2. Anziché scrivere JP NZ, D01H (°) si potrà anche e più co modamente scrivere JP NZ, SU dove SU è la label che avremo provveduto a

---

(°) - È lecita l'omissione degli zeri non significativi.

segnare a fianco dell'istruzione, posta all'indirizzo D01H, alla quale si desidera saltare con la condizione NZ (cioé flag Z = 0).

È parimenti possibile riferirsi ad una label per l'inizio di una subroutine (e scrivere, ad esempio, CALL GIOCONDO, dove GIOCONDO è il nome dell'indirizzo dell'istruzione iniziale della sobroutine).

Come è noto, dal programma *"sorgente"* (ingl. source program), scritto in Assembly, si passa al programma *"oggetto"* (object program) scritto in assoluto attraverso un programma traduttore, detto più propriamente *assemblatore*, scritto naturalmente in linguaggio assoluto. Questa traduzione può essere fatta attraverso un computer o anche tramite un sistema di sviluppo come lo ZDS della Zilog o quello che noi proporremo nel capitolo 3 (NASCOM I).

Diamo qui di seguito le regole fondamentali di Assembly, rimandando, per il set di istruzioni Z80, alle tavole dell'Appendice A in fondo al volume. Dato lo scopo fondamentale di questo libriccino (fornire esempi di programmi) non abbiamo qui la pretesa di essere esaustivi. Su taluni punti comunque coglieremo l'occasione per fare osservazioni non sempre evidentissime in altri manuali.

Formato di una istruzione Assembly

Gli elementi costitutivi e, sotto ciascuno di essi, le varie parti di una istruzione tipica sono riportati qui sotto:

Label	OP-Code	Operandi	Commenti
RIPET	ADD	A, (HL)	Totalizza in Acc.re

La *LABEL* (RIPET nel caso che funge da esempio) è associata all'istruzione, identificandola. Essa, naturalmente può anche mancare. Può essere evocata da una istruzione di salto o da una CALL.

L'*OP-CODE* (codice operativo) è seguito dagli *OPERANDI*, cioè i termini che sono oggetto dell'op-code. Quando questi sono due, sono separati da una virgola, *con il primo operando che costituisce anche il campo del risultato*: nel nostro esempio, il contenuto della cella RAM il cui indirizzo è "puntato" tramite la coppia HL è aggiunto all'Accumulatore. Ossia, se, prima dell'istruzione, l'Accumulatore contiene putacaso 1BH e HL sta puntando all'indirizzo di memoria ODF2H, contenente il dato 12H, dopo l'istruzione l'Accumulatore conterrà: (1B + 12) H = 2DH.

I *commenti*, opportuni per meglio chiarire il senso delle varie operazioni, possono essere scritti sulla destra. I vari elementi del formato debbono essere separati da uno o più blank.

Un punto e virgola dovrebbe precedere i commenti, ma noi, per semplicità editoriale, non lo metteremo mai (sperando che nessuno se ne abbia a male).

Operandi

Possono essere:
- *Registri* A (Accumulatore); B; C; D; E; H; L, nonché I e R (°).
- *Coppie di registri* AF (Accumulatore + Flag register); BC; DE; HL; e *registri doppi* a 16 bit IX; IY e SP. C'è anche la coppia AF', chiamata in gioco dall'istruzione EX AF, AF' (°°).
- *Byte di memoria*, o coppie di byte dette "voci" (word), chiamati in gioco o tramite il loro indirizzo o tramite coppie o indici; come s'è già visto.
- Lo *stato di uno dei 4 Flag* Cy (Carry), Z (Zero), S (Segno), P/V (Parity/Overflow), riassunto della tabella seguente (dei condition code, abbr. cc.):

Flag	stato "on"	stato "off"
Cy	C	NC
Z	Z	NZ
S	M(Minus)	P (Plus)
P/V	PE (Even)	PO (Odd)

Si noti che il flag S indica, quando vale 1, numeri *negativi*, onde P denota la condizione *maggiore o uguale* a zero (e, nel caso di una comparazione, 1° termine ⩾ del secondo).

Label (Lett.: etichetta)

Come si è detto, una label è associata ad un indirizzo o ad un valore numerico, fino ad un massimo di 16 bit. Per scrivere una label occorre rispettare le regole che seguono:
- il primo carattere sia una lettera;
- gli altri caratteri possono anche essere cifre, ma non segni speciali (compreso il blank; sono cioé vietati nomi divisi tipo A CAPO);
- numero massimo di caratteri pari a sei (si possono anche aggiungere caratteri, ma l'Assemblatore li ignora: es. GIOCONDO e GIOCON sono per lui la stessa label).

Esempi di label non corrette: DUE +; 2X. Corrette sarebbero invece DUEPIU (senza accento) e X2.

(°) - I è utilizzato in pratica solo per gli interrupt "vettorizzati" e R, che serve al "refresh" delle memorie dinamiche, non lo useremo mai.

(°°) - Gli altri registri di salvataggio BC'; DE'; HL' sono evocati solo in modo *implicito* dalla istruzione EXX.

Alle label, nel corso di un programma, *non* in sede di definizione, possono essere associate espressioni persino complicate negli assemblatori più potenti (sistemi di sviluppo con alta capacità di memoria e dischetti o addirittura grossi computer). Noi facciamo riferimento ad un sistema di medio calibro e ci limitiamo alla possibilità di aggiungere o togliere, ad una label, una costante e alla possibilità di definire, per una label che designa un valore, un numero negativo espresso con segno, anziché in complemento a 2.
Esempi:

```
LD HL,IND - 2
CALL SUBRTN + 7
```

Se IND si riferisce all'indirizzo CF0, e IND − 2 viene associata all'indirizzo CF0 − 2 = CEE, sito due posti prima. Con SUBRTN + 7 si chiama la routine che inizia 7 byte più avanti di SUBRTN (utile nel caso di subroutine ad entrate multiple).

```
ADD A, - 1AH
LD BC, - VALOR
```

La prima aggiunge ad A il valore *opposto* a 1AH, mentre la seconda istruzione carica nella coppia BC il dato a 16 bit di segno opposto a quello definito da VALOR.(°)

Anche per i *salti relativi* si possono usare le label. Molti assemblatori pretendono l'aggiunta di − $ sulla destra. Se il salto relativo è dato direttamente col numero dei byte da saltare (*effettivi*, senza la sottrazione di − 2, croce e delizia di chi opera in assoluto) occorre far precedere il carattere $.

Esempi:

```
JR    Z, CICLO - $
JR    $ + 5
```

Nel primo caso, il traduttore provvede a calcolare il salto relativo necessario per passare, per Z = 1, dall'indirizzo attuale a quello corrispondente a CICLO.
Con buona pace dei pignoli (e malgrado il fascino della moneta dello Zio Sam) **ometteremo**, per non creare complicazioni editoriali, **questo simbolo**.

PSEUDO CODICI OPERATIVI

Vanno intesi come comandi all'assemblatore piuttosto che all'elaboratore. L'assemblatore riceve l'ordine di compiere determinate operazioni, consistenti nell'inserimento nel programma di opportuni valori.

(°) - Ricordiamo nuovamente che i numeri senza H (o B per i binari puri) vengono considerati decimali e, nel corso della traduzione, sono automaticamente convertiti in binario.

Gli pseudo codici sono, principalmente:

ORG nn

nn è un numero a 16 bit (quattro cifre hex.). Questo pseudo opcode pone il contatore di riferimento indirizzi al valore (iniziale) nn. Fissa, in sostanza, l'inizio del programma.

Questo inizio è spesso, per così dire, "fisico", in quanto, come si vedrà in molti esempi, l'indirizzo del byte contenente la prima istruzione con cui iniziare l'esecuzione spesso è situato dopo aree di lavoro (quali ad esempio quelle definite da pseudo opcode tipo DEFB e DEFW) oppure dopo una o più sobroutine scritte "in testa"

EQU nn

Assegna a una label il valore nn. È il modo più semplice di fissare una costante, definendola con un nome, o un indirizzo. Vale la regola per cui a dato (o indirizzo) diverso deve essere assegnata una diversa label.

I codici che seguono sono da scrivere *a destra di una label*:

DEFB n

n = costante di due cifre hex.(otto bit). Serve a definire il contenuto di un byte, di indirizzo pari al *corrente* valore del contatore di riferimento, uguagliandolo ad n.

Per spiegarsi meglio: l'assemblatore, nella "passata" in cui assegna indirizzi e valori usa un contatore (il cui valore iniziale corrisponde a quello stabilito da ORG nn). Man mano che traduce un'istruzione, il compilatore incrementa tale contatore del numero di byte di quella istruzione. Arrivando ad una istruzione tipo: PINCO DEFB n, assegna al byte di quell'indirizzo cui si è arrivati il valore n ed il nome PINCO all'indirizzo (in modo che, se in seguito è citata la label PINCO, ad essa è sostituito questo indirizzo).

I codici tipo DEFB o DEFW ecc. è opportuno siano posti in testa o in fondo al programma (°). Nei problemi più semplici li metteremo in testa, come pure ORG

DEFB 's'

serve come il precedente, con s che è un carattere tradotto automaticamente in ASCII

DEFW nn

ha funzione analoga ai precedenti, ma per una *coppia* di byte (W = Word) caricati col valore nn in base alla solita regola per cui il byte più significativo

---

(°) - Ponendoli in mezzo, si sarebbe costretti a farli precedere da una istruzione di salto, per evitare che il contenuto di quei byte, contenenti dati, venga interpretato come istruzione.

è caricato a destra di quello più "leggero"

DEFS nn

riserva nn byte di memoria partendo dal valore corrente del contatore di riferimento. In questo caso i valori iniziali *non* sono precisati. Utile per tabelle e simili. La label punta al primo degli nn byte

DEFM 's'

dove s rappresenta un *messaggio* in ASCII (che noi scriviamo in chiaro) di lunghezza massima pari a 63 caratteri.

Vediamo un esempio di come un programma in Assembly può iniziare con degli pseudo opcode. Immaginiamo il *"listing"* che il traduttore fornisce al termine e che comprende, sulla sinistra, gli indirizzi crescenti in cui è scritto il programma ed il loro contenuto in esadecimale.

Indirizzi	Contenuto	Label	Codice Assembly	Commenti
			ORG D00H	
D00	0F	PIPPO	DEFB 15	
D01	34 12	PLUTO	DEFW 1234H	

Come si vede, sulla sinistra degli pseudo opcode compaiono i valori 0F (traduzione esadecimale di 15) nella posizione di memoria D00 e 1234H (si noti l'usuale inversione low order/high order) nei byte D01 e D02. D00 è stato assegnato tramite l'ORG.
Quando, più avanti, è chiamato in ballo, poniamo, PIPPO si ha, ad es.:

```
D17        3A 00 0D        LD A, (PIPPO)
```

del tutto equivalente a:

```
D17        3A 00 0D        LD A, (D00H)
```

che è però meno comodo.

È pure comodo il fatto di poter assegnare un valore ad uno o più byte, *senza* che sia vietato modificarne nel corso del programma il valore (semprechè si tratti di memoria RAM però!). Es., dopo LD (PIPPO), A se l'Accumulatore contiene 1BH anche (PIPPO) avrà tale valore.
Nel caso di un messaggio in ASCII si può avere ad esempio:

Indirizzi	Contenuto	Label	Codice Assembly	Commenti
D15	41 4C 4C 41	MESS	DEFM 'ALLARME'	
D 19	52 4D 45			

L'Assemblatore traduce in ASCII (cfr. TAV. II Cap. 1) e, nel listing, scrive questi codici a 4 a 4 sulla sinistra.

Lo pseudo codice: END

segna la fine del programma (negli esempi lo omettiamo quasi sempre, anche perchè non obbligatorio in certi casi).

Un potente pseudo opcode è, infine, MACRO, che serve a definire con una label un gruppo di istruzioni. La MACRO differisce da una subroutine in quanto non è oggetto di istruzione CALL. Comunque non insistiamo, dati i limiti di questo libro, su tale argomento, bastandoci di averne citato l'esistenza.

CODICI EFFETTIVI (Mnemonici e assoluti)

In quanto segue, ci limiteremo, per i motivi già enunciati nell'Introduzione, a richiamare le cose essenziali, facendo riferimento all'Appendice A. D'altronde, oltre che documentarsi con i testi richiamati in Bibliografia, il lettore potrà approfondire indirettamente la materia in tutti gli esempi che gli saranno sottoposti.
Verranno usate le notazioni seguenti (anche e soprattutto nelle tabelle della *Appendice B):*

b	= un numero (da 0 a 7) indicante singolo bit tra gli 8 di un registro o posizione di memoria;
CC	= condizione di un flag (Z; NZ; C; NC ecc.);
d	= registro a 8 bit (o cella di memoria) *destinazione* del dato;
dd	= registro a 16 bit destinazione del dato;
e	= numero relativo a 8 bit (in cpl. a 2 se negativo);
1	= una tra 8 posizioni di memoria in pagina zero (0, 8, 16, 24, 32, 40, 48, 56);
n	= numero binario a 8 bit;
nn	= numero binario a 16 bit;
r	= registro a 8 bit di uso generico: A, B, C, D, E, H, L;
s	= registro a 8 bit o posizione di memoria *sorgente* del dato;
ss	= registro a 16 bit sorgente del dato;
Sb	= singolo bit in un registro a 8 bit o posizione di memoria.

Si aggiunga:
- "L" denota gli 8 bit meno significativi e
- "H" denota gli 8 bit più significativi di un registro a 16 bit;
- tra parentesi () viene messo un indirizzo (od un label) di una posizione di memoria o di un porto di Input/Output (°).

(°) - Pensiamo di tradurre correttamente con porto l'inglese "port", anzichè, come altri fanno, con porta (che si confonde con la corrente traduzione del termine "gate", circuito logico and, nand ecc.).

Gruppo di caricamento a 8 bit

L'Assembly Z80 per ogni tipo di trasferimento dati usa un unico codice: LD (LoaD), il che è motivo di semplicità e chiarezza. Tutti i possibili tipi di trasferimento a 8 bit sono riassunti nella TAV. A-1. Questa, come altre consimili, porta sulla verticale la destinazione e sull'orizzontale la sorgente, onde ad esempio, volendo il codice assoluto del caricamento di A in D (Assembly: LD D,A) si ha, sull'intersezione della colonna intestata "A" con la riga intestata "D" l'esadecimale 57.
Sono possibili in Assembly i caricamenti che seguono:

- da registro a registro: LD r,r'
  es. LD A,B (assoluto: 78)
- l'immediato n in registro: LD r,n
  es. LD B,2FH (ass.: 06 2F)
- da memoria, tramite (HL), a registro: LD r, (HL)
  es. LD C,(HL) (ass.: 4E)

Se in HL c'è 0C02H e in (0C02H) c'è il valore 12H, in C viene messo tale valore.

- da memoria a registro tramite registro indice IX o IY:
  LD r, (IX + d) o LD r, (IY + d)
  dove d è il *displacement,* in esadecimale, rispetto al valore contenuto in IX o I Y. Es. LD A, (IX + 12H) (ass.: DD 7E 12)

Se in IX c'è l'indirizzo 0D12H, in A è caricato il contenuto del byte di indirizzo (D12 + 12)H = 0D24H.

- trasferimenti inversi ai due precedenti:
  LD (HL), r   LD (IX + d), r   LD (IY + d), r
- l'immediato n in (HL) o (IX + d) o (IY + d):
  LD (HL), n   LD (IX + d), n   LD (IY + d), n
  es. LD (HL), 12 (decimale): in assoluto si ha: 36 OC.
- in Accumulatore tramite (BC) e (DE) (°)
  LD A, (BC) e LD A, (DE)
- in Accumulatore da memoria: LD A, (nn)
  es. LD A, (CFFH) (ass. 3A FF 0C)

Se in posizione di memoria CFFH c'è il dato A2H, esso è messo in A.

(°) - Con i puntatori BC e DE non si può caricare direttamente, come avviene con HL, in registri diversi da A.

- trasferimenti opposti ai precedenti:

LD (BC),A LD (DE),A e LD (nn), A

Tutti questi codici *non* influenzano alcun flag.

Ci sono poi le istruzioni LD A,I e LD I,A per il caricamento in A del vettore di interrupt I e viceversa, il cui uso illustreremo negli esempi dedicati.

Le istruzioni LD A,R e LD R,A non sono praticamente mai usate.

Facciamo due sole osservazioni:

1) si usino sempre solo istruzioni *previste* (per esempio LD E, (BC) non esiste!); per evitare errori del genere, specialmente le prime volte è indispensabile tenere sott'occhio le tavole dell'Appendice A;
2) per le celle di memoria RAM è molto utile l'indirizzamento indiretto tramite (HL). Volendo ad esempio caricare successivamente in A tre byte di indirizzo 200H, 201H e 202H, con LD A, (200H) e, in seguito LD A, (201H) poi LD A, (202H) si impiegano 3 x 3 = 9 byte di programma. Facendo invece inizialmente LD HL, 200H (= tre byte), ci vorranno poi tre volte LD A, (HL) (= 1 byte) e due volte INC HL (per passare alla posizione successiva, istruzione pure di 1 byte), il tutto fa un totale di 3 + 3 + 2 = 8 byte.

Gruppo di caricamento a 16 bit (TAV. A-2)

Si possono avere i seguenti trasferimenti:

- dell'immediato nn in una coppia o doppio registro (indice):

LD dd,nn oppure LD IX,nn o LD IY,nn
con dd = BC, DE, HL o SP (Stack Pointer)
es. LD BC 101AH (assoluto: 01 0A 10)

- di una voce in una coppia o doppio registro:

LD HL,(nn) oppure LD dd,(nn) o LD IX,(nn) o LD IY,(nn)

producono il trasferimento del contenuto di due byte, di cui il primo è all'indirizzo nn in HL (o BC, DE, SP oppure IX, IY) *con il primo in L e il secondo in H* (attenzione!)

es. LD HL,(2130H) (ass.: 2A 30 21)

Se in (2130H) c'è 1EH ed in (2131H) v'è B3H, in HL è messo 31EH.
Si noti come il trasferimento LD HL, (nn) richieda 3 byte, contro i 4 degli altri accoppiati e doppi (°).

(°) Qui e altrove abbiamo preferito raggruppare istruzioni sostanzialmente simili anche se diverse in numero di byte dell'assoluto.

- di una coppia o doppio registro in una voce:

LD (nn),HL   LD (nn),dd   LD (nn), IX   LD (nn), IY

il byte "low order" della coppia o doppio registro (es. L) va in (nn) mentre in (nn + 1) va il byte "hign order" (es. H).

- di HL o IX o IY nello Stack Pointer SP:

LD SP,HL   LD SP,IX   LD SP,IY

- Istruzioni PUSH:

PUSH qq   PUSH IX   PUSH IY

Il contenuto, ad es. di qq (ove qq sta per BC, DE, HL o la coppia AF, cioè Accumulatore più flag register) viene "ammonticchiato" sulla "catasta" (ingl. stack).
Per l'esattezza, se ad es. HL contiene 1234H e SP sta puntando all'indirizzo 2003H, dopo l'istruzione PUSH HL il contenuto di H è caricato nella cella di indirizzo 2002 (dopo che SP è stato decrementato una prima volta), poi SP è di nuovo decrementato assumendo così il valore 2001 e a questo indirizzo è posto 34 H. Questi passaggi sono illustrati nella Fig. 2-1.

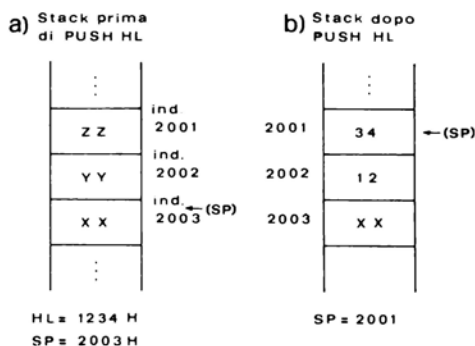



Figura 2-1 PUSH di una coppia di registri sullo stack.

Il PUSH consente il salvataggio, cioè il deposito temporaneo, di dati che servono in un secondo momento, lasciando libera una coppia di registri per altri usi. In un programma reale che utilizzi lo stack, SP deve essere inizialmente posto ad un valore opportuno, solitamente *all'indirizzo più alto di memoria disponibile* (°).

(°) - Più un'unità. Ad es. con 2K di memoria, l'indirizzo più alto è 3FFH. Si porrà inizialmente SP = 400H: il primo PUSH carica nelle celle 3FF e 3FE.

Occorre fare attenzione che:
1) ogni PUSH trovi nel prosieguo del programma il suo corrispettivo POP;
2) non vi siano troppi PUSH consecutivi prima di un POP: si corre il rischio di un "gonfiamento" della catasta quanto mai pericoloso.

Si potrebbero invadere aree di memoria destinate ad altri dati o istruzioni. Se questo avviene su RAM si distruggono dati o addirittura si modifica il progamma (ciò avviene in un microcalcolatore o sistema di sviluppo). Se si invadono aree ROM non succedono tali disastri, ma è il salvataggio previsto dal PUSH a fallire (e col successivo POP si ripristina un dato ... fasullo).

Si fa anche notare che, affinchè le istruzioni PUSH e POP abbiano senso è necessario che nel progetto della macchina a micropocessore sia prevista una area "scratchpad" per i dati, su RAM (nella maggior parte dei casi anche più semplici e ormai usuale 1K di RAM, data la diminuzione dei costi di tali dispositivi).
- Istruzioni POP:

POP qq   POP IX   POP IY

hanno funzionamento ed effetti opposti rispetto alle precedenti e servono al ripristino di valori salvati in precedenza sullo stack.

Con riferimento ancora alla Fig. 2-1, in cui la situazione di partenza è quella illustrata in b), dopo POP HL si ritorna alla situazione in a), con il contenuto di HL = 1234H, anche se in precedenza questo era stato modificato.

Si faccia attenzione a non commettere un errore, comune a molti principianti (noi stessi ci siamo cascati un paio di volte ...): per riottenere nei registri di partenza dati salvati sullo stack *occorre fare i* POP *in ordine inverso.*
Es., dopo PUSH HL PUSH DE PUSH BC, l'ordine corretto per il ripristino è: POP BC POP DE POP HL.
Notiamo infine che nessun load a 16 bit influenza i flag.

Istruzioni di scambio (TAV. A-3)

In Assembly si scrivono:
EX DE, HL; EX AF,AF'; EXX; EX (SP),HL; EX (SP),IX; EX (SP),IY il cui significato dovrebbe essere evidente di per sè (scambio dei contenuti del 1° e 2° operando). Le istruzioni EX AF,AF' e la EXX utilizzano i registri "duplicati" AF', BC', DE', HL' (la EX AF,AF' solo AF' e la EXX gli altri tre). Rappresentano una alternativa ai PUSH e POP di salvataggio. Quelle di esse ad un solo byte sono, perciò, molto veloci e particolarmente adatte nella risposta ad interrupt.

Istruzioni di trasferimento blocchi (TAV. A-4)

Istruzione di ricerca di blocchi (TAV. A-5)

Dalle stesse tavole (e dal testo dello Z80 Technical Manual) si dovrebbe dedurre facilmente il significato di queste istruzioni che sono:

- trasferimento blocchi:	LDI	LDIR	LDD	LDDR
- ricerca di blocchi:	CPI	CPIR	CPD	CPDR

Come si vede, i due gruppi presentano una stretta analogia.
Per utilizzare queste potenti istruzioni occorre preliminarmente porre:

HL = indirizzo 1° byte del blocco da trasferire o da comparare;
DE = indirizzo 1° byte del blocco su cui trasferire (non occorre con le istruzioni di ricerca);
BC = lunghezza dei blocchi (uno solo, con le istruzioni di ricerca).

In tutte le istruzioni dei due gruppi BC è decrementato ad ogni passo, mentre HL (come pure DE nel primo gruppo) sono decrementate nelle istruzioni che comprendono la D (es. LDDR) e incrementate in quelle che possiedono la I (es. CPIR). Nei codici con la R si va avanti automaticamente fino a che BC = 0 (°).

Esempi: LDDR: se BC = 100H HL = 1200H DE = 1400H il blocco di 256 byte fino all'indirizzo 1200 *compreso* è copiato nel blocco di pari numero di byte fino alla cella 1400H.
Con gli stessi valori di BC e HL (qui DE non conta) la CPIR, supponendo che l'Accumulatore contenga 2CH, fa effettuare la ricerca della prima cella RAM di contenuto 2CH a partire dall'indirizzo 1200H in avanti. Appena ne trova una di tale contenuto il ciclo si arresta con l'indirizzo di essa in HL.
Se invece nessuna cella contiene 2CH, il ciclo ha termine con BC = 0 (che dà la condizione "non trovato").

Istruzioni aritmetiche e logiche a 8 bit (TAV A-6)

Le istruzioni ADD, ADC (ADD with Carry), SUB, SBC (SUBtract with Carry), AND, XOR ecc. hanno tutte come riferimento implicito l'Accumulatore, che costituisce il 1° operando e il campo del risultato, quando ve ne sia uno numerico. L'Accumulatore è nominato esplicitamente con ADD:

ADD A,B; ADD A,C ecc. ADD A,(HL); ADD A,(IX + d);
ADD A, (IY + d); ADD A,n;

(°) - Ponendo inizialmente BC = 0 il numero di celle oggetto di trasferimento o ricerca arriva addirittura a 64K,il che è pazzesco se si considera che tale è la massima capacità di memoria indirizzabile con lo Z80!

parimenti è nominato nelle istruzioni omologhe con ADC *e con SBC* (°). Invece col codice SUB l'Accumulatore è sottinteso:

SUB B; SUB C; SUB (HL) ecc. ma SBC A,B; SBC A,(HL) ecc.

Operazioni logiche

L'accumulatore A è *implicito.*

- AND fa *l'and logico, bit per bit,* tra l'Accumulatore e un altro registro o cella (HL) o il dato immediato n:

Esempi: AND C: se C = 23H cioè 00100011B, dopo l'operazione i bit A_7, A_6, A_4, A_3 e A_2 di A sono zero, gli altri restano immutati.

- OR fa *l'or logico,* bit per bit.

Esempi: OR 3 BH, cioè con l'immediato 00111011B (anzi, in casi come questo conviene direttamente scrivere, per maggior evidenza: OR 00111011B). Se A = 10010010, dopo l'operazione A = 10111011, ovvero si conservano solo gli zeri *comuni* ai due operandi.

- XOR fa *OR,* che dà bit risultante 0 quando i bit corrispondenti sono uguali, 1 se sono diversi. Lasciamo il non difficile esempio al lettore, che già sbadiglia.

Si fa notare che non esiste alcuna istruzione per sommare ad A il contenuto di una cella RAM, se non tramite il puntatore HL (o gli indici IX e IY.)

Istruzioni "curiose" ma a volte utili sono quelle di A con se stesso:

- ADD A,A *raddoppia* il contenuto di A o, che è lo stesso, fa uno shift a sinis tra di un bit, caricando un bit zero a destra;
- ADC A,A raddoppia A e aggiunge il Carry;
- SUB A rende A = 0, in quanto si fa A - A;
- SBC A,A fa A −A −Cy, cioè pone −1 (in complemento a 2: ciò vuol dire tutti 1) in A se Cy = 1, altrimenti A = 0. *In altri termini, Cy è copiato in tutti i bit dell'Accumulatore.*
- XOR A opera praticamente come SUB A. Per azzerare A va pure bene LD A,0, per quest'ultima istruzione lascia immutati i flag Cy e S, che invece le altre due resettano.

- Comparazione (CP): CP B ecc. CP n; CP (HL) e le analoghe con IX e IY. Viene fatta una comparazione di A con l'operando menzionato nell'istru-

---

(°) - Ci sembra di non dover insistere troppo sul significato di questi codici e sulla differenza tra operazioni senza o con Cy.

zione e, *senza alterare il contenuto di A,* si attivano i vari flag *come se l'operando fosse stato sottratto da A.*

Esempi: CP 10H: se A = BH il flag S viene settato (essendo A minore di 10H) e il flag Z = 0. Se invece A = 10H si ottiene S = 0 e Z = 1. Nei due casi il flag Cy risulta uguale a 1, nel primo, a zero nel secondo.

- Incremento/Decremento: codici INC e DEC associati ai soliti operandi "source" di TAV. A-6: INC A; INC B; INC (HL) ecc. idem con DEC. Si aggiunge o si toglie un'unità ai diversi registri o alla cella (HL).

I Flag S, Z ecc. sono naturalmente influenzati secondo logica.
A proposito di flag, si ricordi che, quando si hanno dubbi sul modo in cui una istruzione li influenza o meno, si deve consultare l'Appendice B che riporta tali dettagli, per ogni istruzione.

Istruzioni di impiego generale su AF (TAV. A-7)

- DAA aggiusta l'Accumulatore per le operazioni in BCD (impaccato):

  DAA dopo un ADD o SUB ecc. opera la necessaria correzione perchè i due semibyte di A siano in BCD. Es. con A = 29H, dopo ADD A,13H in A si ha 3CH *e non* 42H, che invece si ottiene automaticamente facendo seguire la DAA.

- CPL inverte tutti i bit dell'Accumulatore.
- NEG fa il complemento a 2 dell'Accumulatore.
- CCF (Complementa il Carry Flag): se Cy = 0 lo rende 1, viceversa se Cy = 1.
- SCF (Set Carry Flag): pone Cy = 1. Per il reset di Cy si può usare OR A o AND A (or o and di A con se stesso) che lasciano immutato A, mentre con XOR A si ha contemporaneamente A = 0.

  OR A e AND A, istruzioni apparentemente "oziose" (lasciano A immutato) sono pure *indispensabili* in altri casi molto frequenti, ossia quando si debbono attivare i flag S e Z in conformità ad un valore precedentemente caricato in A. Poichè l'istruzione LD A, s lascia inalterati i valori precedenti di tali flag, *il dimenticare di far seguire OR A* (o AND A) *può essere spesso un errore serio.*

Istruzioni aritmetiche a 16 bit (TAV. A-8)

Il lettore, a questo punto, è senz'altro in condizione di interpretare da solo questa tavola. La destinazione, cioè il campo del risultato, è la coppia

HL (o, in un paio di casi, IX e IY). Si hanno istruzioni del tipo:

ADD HL, BC; ADD HL, DE; SBC HL, DE; INC HL ecc.

Notiamo solo:

1) l'istruzione ADD HL,HL raddoppia il contenuto di HL;
2) non esiste un'istruzione SUB ma solo SBC (SUBtract with Cy). La SUB può essere però surrogata *facendo precedere un reset di Cy;*
3) ADD, ADC e SBC influenzano *solo* Cy (che, se non è ancora stato capito, è il riporto o prestito richiesto dal *bit sedicesimo*); INC e DEC non influenzano neanche Cy;
4) l'istruzione SBC HL, HL fa copiare Cy in tutti e 16 i bit di HL (azzera HL, se Cy = 0).

La sequenza di istruzioni: AND A (rende Cy =0); SBC HL,HL è perciò equivalente a: XOR A seguito da LD H,A e LD L,A o anche a: LD HL,0. Tutte queste sequenze impegnano 3 byte.

Rotazioni e shift (TAV. A-9)

Queste istruzioni operano su dati a 8 bit contenuti nei registri di impiego generale o anche, tramite il puntatore HL o gli indici IX e IY, su qualunque posizione di memoria. I disegni associati alla tavola dovrebbero illustrare con immediata evidenza come operano le varie istruzioni, le cui mnemoniche sono del tipo:

RRC B   RRC C ecc.   SLA A   SLA (HL) ecc.

Nelle rotazioni e shift (= scorrimento, di bit) è pure in gioco il carry Cy il che può servire alla successiva, ciclica estrazione di bit.
Si noti come le rotazioni riguardanti l'Accumulatore sono eseguite più convenientemente con i codici *a un sol byte:* RLCA; RRCA; RLA; RRA anzichè con le analoghe, ma a due byte, RLC A; RRC A; RL A; RR A.
Infine le istruzioni:

RLD (HL) (Rotate Left Digit)
RRD (HL) (Rotate Right Digit)

operano su gruppi di 4 bit e servono nell'aritmetica decimale (BCD).

Esempi: RLCA: se A contiene 10110110, dopo l'istruzione contiene 01101101 con Cy = 1
RRA: applicata allo stesso contenuto di A di sopra, dà, supponendo inizialmente Cy = 1 : A = 1101011, con Cy = 0.
SRA A: con A iniziale come sopra, produce A = 11011011 e Cy = 0 (il bit A_7 - quello del segno! - si conserva e shiftano gli altri).

Manipolazione dei bit (TAV. A-10)

Anche qui il significato e l'uso della tavola dovrebbero esser chiari.
I codici Assembly sono del tipo:

SET b,r RES b,r BIT b,r

o le analoghe con (HL) o gli indici.
Esse consentono, rispettivamente, il *set,* il *reset* o il *test* del solo bit b (b = 7, 6, 5, 4, 3, 2, 1 o 0).
Esempio: BIT 6, D: se il bit 6 del registro D è uguale a zero, il flag Z è = 1 (attenzione a non confondersi!) viceversa, Z = 0 se D_6 = 1.

Va sottolineata la praticità dell'istruzione BIT, rispetto alla tradizionale mascheratura necessaria ad esempio con il micro 8080. Con questa tecnica, anzichè semplicemente BIT 6,D si sarebbe dovuto fare:

LD A,40H (cioè 01000000B) seguito da AND D

per avere il medesimo risultato, modificando per giunta il contenuto di A, cosa che con l'istruzione BIT non avviene. Per contro, quando si opera con più di un bit, possono essere più pratici gli operatori logici: es., per resettare bit 7 e 0 di A basta fare AND 7EH (cioè 01111110B), in luogo di RES 7,A più RES 0,A.

Salti, chiamate e rientri (TAV. A-11)

Queste istruzioni non hanno nulla a che vedere con la ... quadriglia, ma sono importantissime nel gran ballo della programmazione.

I salti possono essere *assoluti, relativi* e anche *indiretti,* cioè tramite (HL) (o gli indici IX e IY). Possono anche essere *incondizionati o condizionati* (sulla base dei cc. Z, NZ, C, NC ecc. già detti).

Salti assoluti: sono del tipo JP nn (salto incondizionato, es. JP C29H o anche JP LOOP) oppure JP cc,nn (es. JP NC, 1000H).

Salti relativi: JR *e* (incondizionato) oppure JCR, *e*; JRZ, *e*; JRR NZ, *e*. Non esistono salti relativi sulle condizioni PO, PE, P e M.

OSSERVAZIONE IMPORTANTE: quando si fanno comparazioni o si fa il test del risultato di un'operazione aritmetica (esempio più classico la sottrazione) le condizioni *"maggiore o uguale"* da un lato e *"minore"* dall'altro possono essere saggiate con i cc. P o M se si lavora con numeri *relativi,* cioè il cui bit 7 è il bit del segno. Se invece si opera con numeri senza segno, il cui "range" va da 0 a 255 (cioè FFH) i codici condizione corretti sono *NC per maggiore o uguale e C per minore.* Bisogna stare attenti a non fare confusione. Vediamone un esempio:

Si confrontino, nell'ambito dei numeri, presi in modulo, da 0 a FF, i dati F4H e 23H: la comparazione (o la differenza) dà C = 0, cioè *No Carry*, mentre invece, dato che il bit 7 nella differenza (esplicita o implicita, nella CP) *si conserva* il flag, risulta pari a 1 e si ha la condizione M.

Solamente quando ci muoviamo nell'ambito dei numeri aventi il bit 7 nullo i cc. NC e C sono perfettamente equivalenti ai corrispondenti P e M (lo stesso avviene quando entrambi i dati hanno il bit 7 unitario: comparando ad es. E5 e D4 si ha NC e, insieme, P, sia che si considerino i due dati come 229D e 212D, vuoi che li si prenda come i negativi −27 e −44, rispettivamente). Inutile dire che la scelta del "range" entro cui muoversi dipende dal programmatore (e dal tipo di problema).

Il numero *e* corrisponde a quello dei byte da saltare, positivo per i salti in avanti, negativo per quelli indietro. In assoluto occorre togliere 2, cosa non difficile con i salti avanti (es. JR $ + 5; si traduce in: 18 03), mentre si complica un poco per quelli a ritroso (°).

Per comodità i valori decimali fino a − 30 di *e* sono tradotti in corrispondenti valori *e* − 2 nella TAV. 2-1.

TAV. 2 - 1
Salti negativi e-2

F (Prima cifra)	Seconda cifra	E (Prima cifra)
—	F	15
0	E	16
1	D	17
2	C	18
3	B	19
4	A	20
5	9	21
6	8	22
7	7	23
8	6	24
9	5	25
10	4	26
11	3	27
12	2	28
13	1	29
14	0	30

I numeri da 0 a 14 (segno meno sottinteso) hanno la prima cifra F, quelli da 15 a 20 hanno E. La seconda cifra è indicata nella colonna centrale. Es. volendo l'assoluto corrispondente all'Assembly JR NZ, —12, si ottiene: 20 F2.

(°) - Esso va contato a partire dal byte che rappresenta l'op-code *escluso* (includendo, nei salti avanti, i byte degli operandi) della istruzione di partenza, fino all'op-code *incluso* dell'istruzione di arrivo.

I salti relativi sono molto utili in quanto: 1) occupano ciascuno 2 byte contro i 3 degli assoluti; 2) un programma comprendente solo salti relativi (e, quindi, in particolare, nessuna subroutine ....) è completamente *rilocabile* (cioè si può pari pari spostare, senza modifiche, in altre parti di memoria).

L'istruzione DJNZ *e* usa il registro B come decontatore, lo decrementa ogni volta e, se $B \neq 0$, fa un salto relativo di *e* byte. Istruzione molto utile per far eseguire un loop un numero di volte pari al valore inizialmente impostato in B.

Salti indiretti: JP (HL); JP (IX) e JP (IY) fanno saltare all'indirizzo inrizzo indicato nella coppia di registri HL o nei registri indice. Si possono impiegare come *deviatori* di programma (questo viene spostato a seconda del valore precedentemente impostato in HL o negli indici).

Chiamate di subroutine: CALL nn o CALL cc, nn (qui la chiamata aviene viene sulla condizione cc: es. CALL NZ, ROUT, si va all'esecuzione di ROUT solo se Z = 0, altrimenti si passa all'istruzione seguente.

nn è l'indirizzo iniziale della subroutine ed è trasferito nel Program Counter PC, mentre automaticamente l'indirizzo della istruzione *successiva* è posto sullo stack.

Rientro da subroutine: RET e RET cc rimettono nel PC, prelevandolo dallo stack, il valore ivi salvato, consentendo la ripresa del programma chiamante (°). Si osservi che le istruzioni CALL e RET agiscono sullo stack e sul suo puntatore SP come le PUSH e POP, rispettivamente. Attenzione quindi alle possibili interferenze reciproche.

Le istruzioni infine RETI e RETN devono *obbligatoriamente* terminare le subroutine di risposta agli interrupt mascherabili, e rispettivamente, non mascherabili.

Gruppo "RESTART" (TAV. A-12)

Il formato è: RST l con l = una tra le dette 8 diverse posizioni di memoria in pagina zero. Consente di eseguire, con un solo byte di codice, la chiamata di una routine sita in pagina zero. Usata per chiamare routine di monitor o simili.

(°) - Questo, nei casi più semplici, è il programma principale o MAIN, ma può anche trattarsi di subroutine di livello maggiore (le routine possono essere *"nidificate"* una nell'altra come scatole cinesi).

Istruzioni di Input e Output (TAVV. A-13 e A-14)

Comandano l'immissione o l'emissione, sul Data Bus, da o verso uno dei 256 "porti" indirizzabili dalla CPU Z80 (e, tramite essi, consentono lo scambio dati con una periferica).
Si hanno le istruzioni:

- IN A, (n) OUT A, (n) n = INDIRIZZO DEL PORTO
- IN r, (C) OUT r, (C)

queste ultime consentono l'indirizzamento indiretto tramite il registro C col dato in partenza o in arrivo sito in qualunque registro di impiego generale r (°).

- INI INIR IND e INDR
- OUTI OUTIR OUTD e OUTDR

Queste istruzioni hanno la funzione di trasferire blocchi di byte (fino a 256) da e verso l'esterno ed operano con un meccanismo analogo alle LDI, LDIR ecc. Il porto è indirizzato indirettamente tramite C, la coppia HL punta al blocco in memoria, mentre come decontatore è utilizzato il registro B.

Controlli della CPU (TAV. A-15)

Si tratta di comandi alla unità centrale:

- NOP (No Operation): la CPU non fa ... nulla e passa alla istruzione seguente.
  Apparentemente banale, può avere varie funzioni: essere inserita in un loop di attesa; sostituire una istruzione che si è preferito cancellare, senza dover riscrivere tutto; a volte, il programmatore accorto riserva in certi punti "critici" tre NOP: quando, in fase di messa a punto, si accorge di dover inserire altre istruzioni, può porre al posto dei 3 NOP la chiamata di una subroutine (che potrà scrivere in fondo alle istruzioni già scritte), che comprende i codici da inserire. È un rattoppo, ma è sempre meglio che dover rifare tutto, specie se il programma è molto lungo.
- HALT blocca la CPU fino a quando non interviene o un comando esterno di reset o di interrupt.
- DI (Disable Interrupt) ed EI (Enable Interrupt) servono, rispettivamente, quando si vuol inibire o consentire la risposta all'interrupt "mascherabile" (cioè, appunto, disabilitabile: con quello non mascherabile non c'è nulla da fare)

(°) - Utile ad esempio quando si voglia una stessa routine di servizio per periferiche diverse (prima della CALL, si carica C dell'indirizzo opportuno).

Infine i comandi IM0, IM1 e IM2 pongono i modi 0,1 e 2 di risposta agli interrupt.

Nella Appendice B sono riportati tutti i codici Assembly mnemonici, con i relativi equivalenti assoluti, gli effetti di ogni istruzione sui vari flag, nonchè i cicli di macchina ed il numero di stati T (cioè di periodi di clock) necessari per l'esecuzione completa. Il *tempo di esecuzione* in microsecondi si ottiene dividendo per la frequenza dell'orologio, espressa in MHz.

Esempio: un'istruzione tipo LD r,n prevede 7 stati T. Se la frequenza del clock è di 2,5 MHz, si ricava:

$$t_{sec.} = 2{,}8 \text{ microsec.}$$

L'ASSEMBLER DEL POVERO

Può capitare a chi non ha troppi mezzi di dover cominciare con un sistema di sviluppo o "personal computer" sul quale non è possibile tradurre dall'Assembly language. In queste condizioni non occorre disperare, ma armarsi di pazienza e di metodo. Lavorare direttamente in linguaggio macchina è possibile, ma in seguito non si riesce più a leggere bene il programmino più semplice.

Consigliamo allora i seguenti passi (successivi all'eventuale stesura di un flow-chart):

1) scrivere il programma con le mnemoniche Assembly sulla destra del foglio, limitando le label a quelle soltanto che necessitano per denominare le destinazioni dei salti e l'inizio di subroutine (inutile dire che qui gli pseudo codici ORG, EQU ecc. non hanno senso);

2) scelta la prima cella di memoria con cui parte il programma, sulla sinistra si scrivono gli indirizzi dei successivi byte e, accanto i codici corrispondenti alle istruzioni Assembly (aiutandosi con le tabelle dell'Appendice A, più comode, a tal fine, che non l'Appendice B); quando si ha un salto ad una istruzione non ancora scritta, lasciare in bianco il byte o la coppia di byte in cui andranno in seguito scritti i valori del salto relativo o assoluto, *tenendoli naturalmente presenti* nel conteggio dei byte di indirizzo memoria, sulla sinistra (in questo conteggio, non dimenticarsi *mai* che dopo 9, in esadecimale, viene A e non 10!!);

3) completare, al termine del programma e relativa traduzione "manuale" i salti lasciati in bianco. Lo stesso discorso vale per le subroutine, ma se si ha l'accortezza di scriverle *in testa* al programma, il problema non si pone, perché quando, nel MAIN, si ha una CALL l'indirizzo è già assegnato.

A questo punto consigliamo l'uso di una tabella inversa a quella della Appendice B (per ogni op-code assoluto, l'equivalente mnemonica): procurarsela o ... costruirsela da bravo bricoleur (è un noioso ma utile esercizio per arrivare a conoscere meglio che si può il set di istruzioni dello Z80). Si verifichino le corrispondenze simbolico/assoluto, si ricontino pazientemente tutti gli indirizzi di memoria (°), i salti e ... pregare il proprio santo personale che alla prova il programma "giri" come si è previsto.

Repetita iuvant. A costo di essere noiosi, vogliamo ribadire un concetto importante a proposito di uso di label. Questa generalmente corrisponde a un indirizzo, assegnato ad esempio tramite DEFB o DEFW. Quando si invoca la label, diciamo così, nuda e cruda, è l'indirizzo che è in gioco *col suo valore*. Se invece si desidera il *contenuto* del byte o della voce che ha quell'indirizzo, la label *va messa tra parentesi:*

Es. ODOF ETICH DEFW 2EA3H

con l'istruzione LD HL, ETICH in HL è caricato ODOFH, mentre con l'istruzione LD HL, (ETICH) in HL è posto 2EA3H (sempreché nel frattempo il programma non abbia modificato il contenuto della voce) (°°).

Esempio applicativo: moltiplicazione per somme ripetute (si riveda il flow chart della figura 1-5).

Trattandosi di una routine, conviene utilizzare registri. Lo facciamo nel modo che segue: il moltiplicando MDO lo immaginiamo in E; il moltiplicatore MRE in B. Per il prodotto PR immaginando di operare con numeri senza segno, il cui "range" arriva a FF (cioé 255 decimale), occorre una *coppia* di registri. Infatti 255 × 255 = 65025 = FEO1H, per contenere il quale occorrono *due* byte operazioni così dette "in doppia precisione"). Si utilizzerà la coppia HL. L'istruzione più adatta per eseguire la ripetuta aggiunta di MDO e PR è: ADD HL, DE (un solo byte contro i più necessari operando con aritmetica a 8 bit). A tal fine però non bisogna dimenticarsi di azzerare preliminarmente il registro D, in quanto MDO e contenuto ovviamente solo in E.

(°) - Gli indirizzi che si scrivono effettivamente sulla sinistra sono relativi al primo byte di ogni istruzione. Es. JP DOOH in assoluto è: C3 OO OD. Se a sinistra di essa è stato scritto l'indirizzo, mettiamo, CFE, a fianco dell'istruzione della riga seguente porremo: CFE + 3 = DO1.

(°°) - Nei flow chart invece non useremo di norma tale distinzione, intendendo ivi indicare con un nome-di-dato il contenuto di un campo, registro o posizione di memoria che sia. Attenzione quindi a non far confusione.

Fatte queste premesse, si ottiene:

Programma P. 2-1

Indirizzi	Contenuto	Label	Codice Assembly	Commenti
			ORG 100H	
100	AF	MOLTIP	XOR A	A = zero
101	57		LD D,A	si azzerano D
102	67		LD H,A	e la coppia
103	6F		LD L,A	HL
104	BB		CP E	MDO = 0?
105	C8		RET Z	Se SI rientra
106	B8	CONFR	CP B	MRE = 0?
107	C8		RET Z	Se SI rientra
108	19		ADD HL,DE	PR = PR + MDO
109	05		DEC B	MRE = MRE − 1
10A	18 FA		JR CONFR	salto relat. di − 4

Il programmetto, con i suoi bravi commenti, segue così da vicino il flow di Fig. 1-5 da non meritare un discorso sul suo procedere passo passo.

Facciamo ugualmente alcune osservazioni (per giustificare scelte fatte ecc.).

L'azzeramento iniziale di A è fatto con due scopi: 1) l'azzeramento, tramite i caricamenti di A in D, H e L di questi registri; 2) le comparazioni nei dati in E ed in B con zero. Chi legge non stenterà a rendersi conto come, operando in tal modo, si sia conseguito il massimo risparmio di byte.

Si noti l'utilità delle istruzioni di rientro condizionato RET Z e il fatto che, in una routine: a) vi possono essere più rientri (già visto in Fig. 1-5); b) l'istruzione di indirizzo più basso non necessariamente è una RET (naturalmente dovrà essere, come in questo caso, un salto incondizionato).

Infine, per il salto relativo JR CONFR, oltre a ripetere che si dovrebbe generalmente scrivere, a rigore, JR CONFR-$, il lettore controlli il salto relativo a ritroso di −4 e la sua codifica con FA, consultando la TAV. 2-1.

Come opera un MAIN che utilizza una tale routine? Immaginiamo di avere MDO in (200H), MRE (201H) e di voler porre PR nella voce formata dalle celle (202H) e (203H) il programma è alla pagina seguente.

Si noti il PUSH di HL prima di chiamare MOLTIP: occorre salvare il valore del puntatore (203H), in quanto MOLTIP ha bisogno di HL per operare la moltiplicazione. All'uscita di essa, con POP HL si ripristina il puntatore, prima però con la EX DE, HL il prodotto calcolato messo in salvo in DE

(e in HL va DE, che tanto non serve più). Le ultime tre istruzioni provvedono allora a caricare in memoria i due byte di PR.

Contenuto	Label	Codice Assembly	Commenti
21 00 02		LD HL, 0200H	carica punt. HL = 200H
5E		LD E, (HL)	MDO in reg. E
23		INC HL	HL =201H (punta a MRE)
46		MD B, (HL)	MRE in reg. B
23		INC HL	HL punta a PR
E5		PUSH HL	salva HL
CD 00 01		CALL MOLTIP	esegui rout. all'ind. 100H
EB		EX DE, HL	PR da HL in DE
E1		POP HL	ripristina puntatore
72		LD (HL), D	carica primo byte di PR
23		INC HL	HL = 203
73		LD (HL), E	carica 2° byte di PR

....

Il lettore che non sia, per precedenti letture, già esperto di questi primi esempietti è invitato a seguirli con grande pazienza e scrupolo passo per passo, consultando le tavole, provando proprie alternative eccetera.

È un lavoro noioso, ma inevitabile. Al termine di questo lungo tunnel di sofferenza, potrà vedere la luce di applicazioni più divertenti e interessanti (almeno così speriamo).

CAPITOLO 3

IL MICROCOMPUTER NASCOM - I PRIMI ESEMPI

Parlare di un microprocessore in generale, senza entrare concretamente in dettagli relativi ad un sistema determinato, non ha molto senso pratico. Non per caso, buona parte dei testi orientati didatticamente si riferiscono infatti ad un qualche sistema di sviluppo e/o "personal computer" che, oltre che per "giocare" ed imparare servono anche, quando hanno un minimo di professionalità, per lavorare seriamente al debug di programmi destinati alla implementazione di prototipi di macchine più o meno complicate a microprocessore.

Anche noi pertanto facciamo riferimento, in molti dei programmi iniziali, anche di media complessità, ad uno specifico microcomputer, il NASCOM-I, prodotto dalla Casa inglese Nascom Microcomputers.

Chi dispone di un sistema diverso ma sostanzialmente consimile contiamo possa ugualmente trarre vantaggio da questo e dagli altri capitoli che ad esso si riferiscono.

Descriviamo quindi le caratteristiche essenziali della macchinetta.

Schema a blocchi (Fig. 3-1)

Il NASCOM-I è un "personal computer" le cui caratteristiche sono tuttavia molto simili a quelle di un sistema di sviluppo di tipo industriale.

Lo schema a blocchi di figura 3-1 mostra, oltre naturalmente alla CPU Z80, i seguenti dispositivi:

- *i blocchi di memoria EPROM e RAM;*
- *dispositivi di decodifica indirizzi* (circuiti multiplexer), le cui uscite vanno agli ingressi "Enable" (rappresentati dalle frecce "E" sullo schema) dei vari chip (memorie e dispositivi di I/O);
- *tastiera* (keyboard) alfanumerica con barra spaziatrice e vari comandi; con scansione a matrice parte hardware (tramite un circuito contatore-decoder), parte software tramite il porto 0 di I/O (°);

(°) - Per i limiti che ci siamo imposti, dobbiamo forzatamente sorvolare sui dettagli tecnici. Di una tastiera con scansione totalmente software parleremo più avanti in un esempio applicativo.

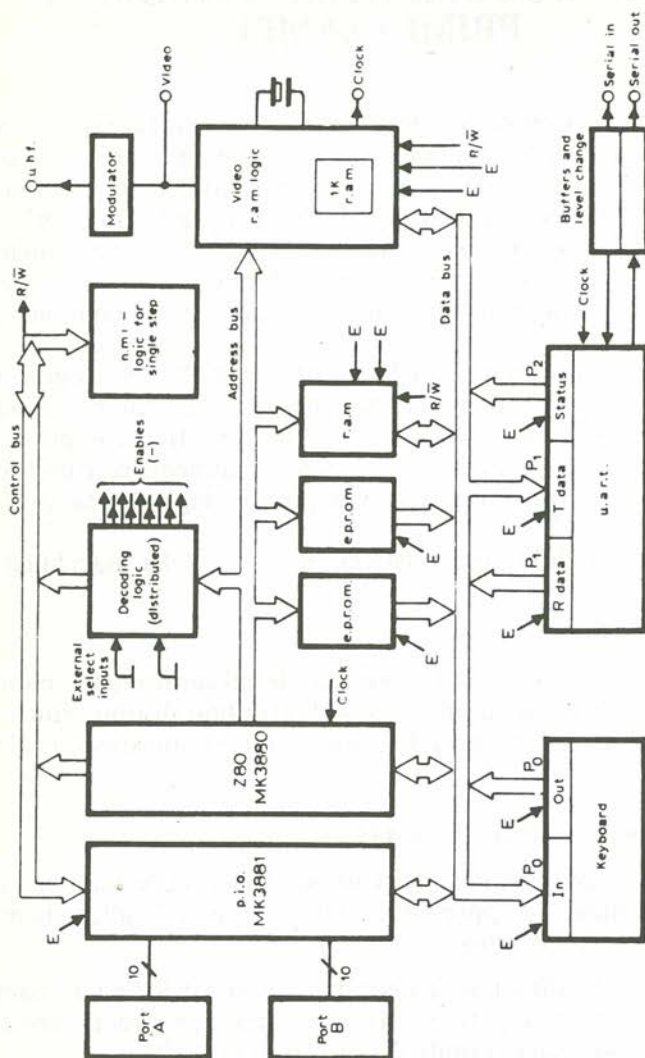



Figura 3-1 Schema a blocchi del NASCOM I.

- *una unità seriale di I/O (UART=* Universal Asynchronous Receiver Transmitter) che è un'interfaccia periferica che serve alla trasmissione/ricezione di dati che arrivano in forma seriale (cioè un bit alla volta; asincrono vuol dire che si accettano anche bit che non arrivano in sincronismo con un clock); questa UART è prevista per consentire l'allacciamento di una unità a cassette magnetiche da impiegare come "mass storage" (memoria, esterna, di massa) per registrare programmi già collaudati, senza dover ogni volta introdurli manualmente da tastiera; serve anche, quando si aggiunge al sistema (che qui stiamo descrivendo sostanzialmente nella configurazione minima) una scheda di espansione di memoria RAM (di 16 kilobyte o più), per caricare da cassetta il compilatore Assembler; l'UART utilizza i porti 1 (per i *dati*) e 2 (per lo "*Status*": l'UART è infatti un dispositivo programmabile come la PIO, la SIO et similia);
- *una unità di I/O parallelo* (PIO): utilizza i porti 4 e 5 per i dati e 6 e 7 per il controllo dei porti 4, e rispettivamente, 5 della PIO: prevista per l'interfacciamento con altre schede, il che tra l'altro consente vari esperimenti per il collaudo anche fisico di sistemi di controllo;
- *display video*: diversi circuiti: modulatore video, nonchè circuiti logici di interfacciamento con una memoria RAM, detta *RAM video*, di un kilobyte di capacità consentono l'utilizzazione, come unità visualizzatrice, di un video monitor o anche di un comune apparecchio TV domestico (disturbi delle trasmittenti private viciniori permettendo); il sistema funziona in modo tale che il modulatore e gli altri circuiti provvedono a mantenere visualizzati in permanenza sullo schermo 16 righe di 48 caratteri ciascuna: per far comparire un determinato carattere in una di queste posizioni video basta caricarlo nella cella corrispondente della video RAM;
- *circuiti logici per il funzionamento in "single step"*, connessi al Control Bus in modo da sfruttare l'interrupt non mascherabile (la cui linea non è pertanto direttamente disponibile per l'utente, il che però non è affatto una seria limitazione); come si sa, il funzionamento "single step" significa l'esecuzione *passo-passo* di un programma ed è una delle più tipiche operazioni di "debugging";
- *orologio (clock) a 2,5 MHz* circa per il pilotaggio della CPU (la quale, come è noto, può operare alla frequenza massima di 4 megahertz).

Per la cronaca, i dispositivi CPU e PIO sono quelli prodotti dalla MOSTEK ("seconda sorgente" dei prodotti Zilog, mentre da poco in Europa un'altra "sorgente" di tali prodotti è l'italiana SGS - ATES), recanti rispettivamente la sigla MK 3880 e MK 3881).

I dati provenienti dalla keyboard vengono caricati *sull'Accumulatore in ASCII code.*

La tastiera ha tutte le 26 lettere alfabetiche, le 10 cifre decimali e i segni speciali correnti, più la barra spaziatrice ed altri comandi (v. Fig. 3-2): *shift*, per dare i segni indicati superiormente in certi tasti.

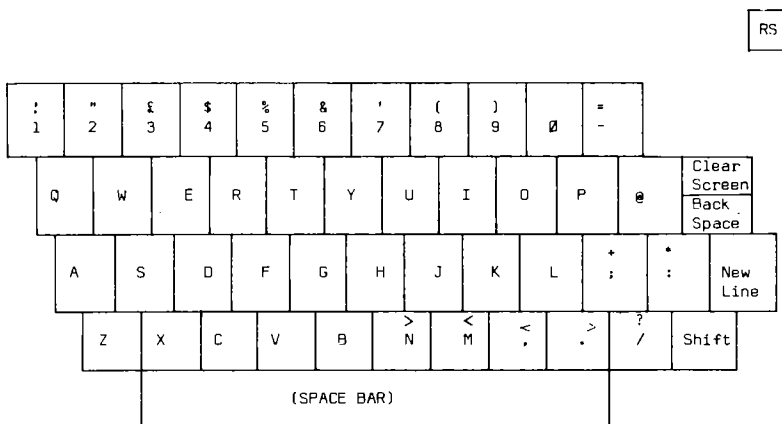



Figura 3-2 Disposizione tasti della keyboard.

(es. & anzichè 6); *backspace*, per il ritorno indietro del *cursore*: questo è un trattino che compare sullo schermo in corrispondenza della posizione di video RAM sulla quale si sta per scrivere; il comando backspace automaticamente annulla (e "abblenca") la posizione precedente; *New Line*, per il passaggio alla riga successiva. A parte c'è inoltre il tasto RS (*Reset*) connesso alla linea omonima della CPU. Come si sa, esso interrompe con priorità assoluta ogni istruzione in corso e fa saltare all'indirizzo zero, dove è collocata la prima istruzione del *sistema operativo* o *monitor.*

Infine, premendo simultaneamente i comandi backspace e shift si ottiene la cancellazione di tutto lo schermo (con l'inserimento nelle posizioni della video RAM di caratteri "blank").

Mappa della memoria (Fig. 3-3)

Partendo dal basso (indirizzi minori) si hanno 4 blocchi da 1 kilobyte ciascuno e cioé:

- 1 k-byte - dall'indirizzo 0000 a 03FF, in cui è collocato, su EPROM, il *monitor* del sistema, denominato NASBUG;

- 1 K dal byte 0400 al byte 07FF, vacante nella configurazione minima del sistema, ma disponibile per una espansione del monitor o simili (es. "supermonitor" da 2K oppure interprete "tiny Basic" o anche serie di routine fisse stabilite dall'utente per propri scopi);
- 1 K dal byte 0800 a quello di indirizzo 0BFF per la video RAM;

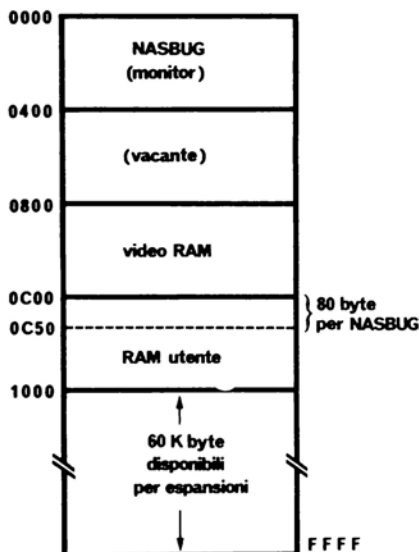



Figura 3-3 Mappa della memoria del NASCOM I.

- 1 K da 0C00 a 0FFF per la RAM utente minima; di queste 1024 posizioni, le prime 80 sono impiegate dal NASBUG, fino all'indirizzo 0C4F, onde l'utilizzatore di norma è meglio che *eviti di iniziare i suoi programmi da indirizzi inferiori a 0C50.*

I rimanenti 60 K sono espandibili utilizzando apposite piastre (come si è detto l'espansione minima per l'assemblatore è una scheda di 16 K).

Video RAM

Gli indirizzi delle posizioni visualizzate sulle 16 righe di 48 caratteri dello schermo corrispondono a quelli indicati nella Fig. 3-4.

La *top row,* cioè la riga che appare in testa, comprende gli indirizzi da 0BCA a 0BF9. Questi, si fa osservare, sono *successivi* a quelli della riga che compare in fondo e viene mantenuta fissa dal monitor, mentre le altre 15 sono interessate allo *"scroll"* (lett.: srotolamento), che avviene così: il NASBUG scrive la prima riga in basso, poi, una volta completatala, la sposta in

alto ecc., in modo che la scrittura procede sempre sulla riga ultima. Dopo quindici di tali passi, naturalmente lo scroll fa sparire la più alta, caratterizzata dagli indirizzi 080A fino a 0839 (°).

Sono invisibili 256 posizioni di video RAM, 16 accanto a ciascuna riga, e precisamente: le 10 posizioni iniziali da 0800 a 0809 accanto alla prima riga

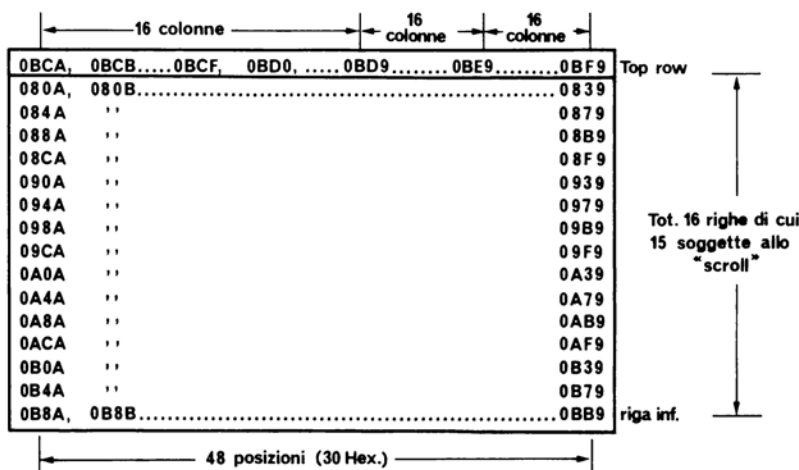



Figura 3-4 Posizioni di memoria della video RAM (parte visibile).

dopo la top row, altre 6: 0BFA ÷ 0BFF, a destra di questa e 15 gruppi di 16 posizioni interposti tra le altre (es. da 083A a 0849 tra la prima e la seconda, dopo la top row).

L'apparecchio TV è pilotato tramite una ROM generatrice di caratteri che fa apparire, per ogni configurazione esadecimale di un byte di video RAM, una rappresentazione visiva di esso. Non riportiamo per ragioni di spazio la relativa tabella, di cui naturalmente gli utenti del NASCOM I dispongono nella documentazione allegata. Diciamo solo che le cifre, le lettere ed i caratteri speciali principali corrispondono quasi tutti all'ASCII code, inoltre sono disponibili altri simboli particolari (frecce di vario orientamento ecc.).

Per spiegarci meglio: se nella posizione video RAM di indirizzo 09EA è contenuto in un certo momento l'esadecimale 46H, su un punto pressochè centrale dello schermo compare la lettera "F" (cfr. TAV. 1-III).

(°) - Tutte queste funzioni sono molto facilitate dalle potenti istruzioni di trasferimento blocchi tipiche dello Z80.

Principali caratteristiche del NASBUG

Comandi

Battendo il tasto RS, il controllo passa al sistema operativo. In tutte le macchine del genere è questo un insieme di sottoprogrammi, interconnessi logicamente tra di loro, che provvedono a varie funzioni relative alla gestione delle periferiche (unità video, tastiera, cassette magnetiche, PIO) e alle operazioni di "debug" per le quali il NASBUG è stato progettato.

Anche in un sistema di controllo un po' complesso, specie se prevede più compiti o, come si dice, "multitask" è necessario un sistema di gestione (chiamato anche con altri termini, come ad esempio "supervisore") che oltre a provvedere alle necessarie inizializzazioni dell'impianto all'atto dell'accensione, gestisca secondo opportune priorità i vari controlli, allarmi, interruzioni ecc. Un simile argomento chiaramente esula dagli scopi del presente manuale, peraltro questi cenni sul NASBUG possono servire a fornire una sia pur pallida idea della materia.

Nel nostro caso il NASBUG, dopo opportune inizializzazioni (valore dello Stack Pointer SP posto pari a 0C33, che è il "fondo" di un piccolo stack proprio del monitor e diverso, come è giusto, da quello dell'utente; pulitura dello schermo, valori iniziali assegnati a proprie aree di lavoro), si mette, per così dire, "in stato di attesa" di comandi operativi provenienti dalla tastiera.

Quando uno di questi arriva, viene decodificato ed eseguito (la tecnica è semplice: ad un certo carattere è associata la chiamata di una opportuna subroutine). Se ad esempio si tratta del comando E (= *"Execute"*) seguito dalla posizione di memoria iniziale del programma dell'utente, il controllo passa a questo (viene forzato nel PC l'indirizzo che segue "E"), prima però il NASBUG provvede a ripristinare in SP il valore dell'*user Stack,* cioè 1000H, che è in fondo alla RAM utente.

Entrare in tutti i dettagli del monitor richiederebbe troppo tempo. Qui ci limitiamo a elencare i principali comandi che il NASCOM accetta, invitando coloro che dispongono dell'apparecchio e della documentazione di corredo ad approfondire le note ed il listing relativi al NASBUG come utilissimo ed istruttivo esercizio. Comunque, quanto stiamo per illustrare è sufficiente alla comprensione degli esempi di programmi che faremo.
Tutti i comandi sono seguiti dalla battuta del tasto NL (cioè New Line).
I principali sono:

- **M** *seguito da un indirizzo:* produce l'evidenziazione sullo schermo del contenuto della cella chiamata e l'utilizzatore (se essa è su RAM) può

modificarne il valore; serve a caricare i successivi byte di un programma (°);
- **S** *più indirizzo* iniziale di un programma (ma in seguito basta battere il solo tasto S o solamente NL): serve al funzionamento *single step:* viene eseguita una istruzione solamente, poi lo Z80 viene tenuto in stato di attesa; sullo schermo compaiono, nell'ordine, i contenuti dei registri SP - PC - AF - HL - DE e BC. Così si possono seguire, ad ogni passo, le modifiche che il programma fa su tali registri; si tratta della più comune e più significativa istruzione di "debugging", utile specialmene in fase di apprendimento;
- **B** *più indirizzo:* serve ad inserire in un programma in prova un *"break point":* in tal caso, dopo il comando E, il programma viene eseguito normalmente ma, arrivando all'indirizzo di break point (semprechè il programma non abbia errori tali che a tale punto non si possa giungere), l'esecuzione si arresta e sullo schermo si hanno i registri già detti, come col comando single step; serve ad una verifica più rapida di quanto non avvenga con il single step, naturalmente occorrerà inserire il break point in un punto significativo e critico; per far ripartire il programma si batte il tasto E.

Altri comandi consentono la tabulazione sullo schermo di blocchi di memoria, il trasferimento di programmi verso e da cassetta (se l'unità a nastro magnetico è connessa) ecc.

Infine il comando "." (punto) seguito da NL serve a terminare la scrittura di un programma (o qualunque altra operazione sia in corso) facendo ritornare il monitor in attesa di altri tipi di comando: usualmente serve, al termine della scrittura di un programma, per passare alla sua esecuzione "single step" o completa.

Per vedere più in dettaglio come si opera, riferiamoci al programma P. 2-1, l'unico che finora abbiamo visto. A tal fine, riportiamo per maggior comodità il listing degli indirizzi e dei soli codici assoluti, aggiungendo in coda alla subroutine MOLTIP il pezzo di MAIN già visto.

Occorrono un paio di modifiche: gli indirizzi, anzichè partire da 100H, li faremo partire da D00H. Ciò è indispensabile, dato che 100H è una posizione ROM del monitor.

Per lo stesso motivo, anzichè le posizioni di memoria 200 e seguenti, useremo le posizioni E00 e successive. In definitiva si ha:

(°) - Dopo la prima volta che M è battuta, ad ogni introduzione di un dato, seguita da NL, o da barra spaziatrice, viene fornito il contenuto della cella successiva, senza che si debba ribattere M.

Indirizzi	Contenuto	Indirizzi	Contenuto
D00	AF	D0C	21 00 0E (carica E00H in HL)
D01	57	D0F	5E
D02	67	D10	23
D03	6F	D11	46
D04	BB	D12	23
D05	C8	D13	E5
D06	B8	D14	CD 00 0D (chiama rout. D00)
D07	C8	D17	EB
D08	19	D18	E1
D09	05	D19	72
D0A	18 FA	D1A	23
(segue sulla destra)		D1B	73
		D1C	76 (comando HALT)

Si notino: la modifica della CALL MOLTIP con CD 00 0D, all'indirizzo D14 e la terminazione del "main" con l'istruzione 76 (HALT).

Vediamo i passi principali del "debug" di questo programmino (°).

Dopo il comando: M E00 seguito da NL per caricare in E00 ed E01 e valori di prova qualsivoglia di MDO e MRE, si fa "." NL, poi M D00 NL.

Comparirà il contenuto di tale cella, che non interessa, quindi scriveremo AF (seguito, a piacere, da NL o barra spaziatrice) poi 57 ecc. fino al codice 76, l'ultimo del programmino. In questa fase di caricamento il monitor ci fornisce automaticamente gli indirizzi successivi, sicché avremo la possibilità di controllarne l'esattezza (anzi, si potrebbe anche utilizzare il NASCOM stesso per scrivere direttamente i programmi, senza la noia del calcolo degli indirizzi successivi). Dopo il "." NL, vogliamo eseguire il single stepping. Si batte S D0C NL e immediatamente sullo schermo compaiono i contenuti di SP (= 1000, come si è già detto), di PC (=0D0F, cioé l'istruzione *successiva* a quella che è *già* stata eseguita), di AF (valore "dont'care", cioé ancora non interessa), di HL = 0E00, messovi dall'istruzione 21 00 0E. DE e BC sono pure valori qualsiasi.

Si batte NL e si ottiene la visualizzazione di SP ancora = 1000, PC = 0D10, con gli altri registri immutati tranne DE, di cui, la parte relativa a E contiene il dato della cella E00. Ancora NL: si ha la modifica, oltre che di PC = 0D11, del solo HL = 0E01 (in seguito a 23. cioé INC HL).

(°) - A titolo di curiosità: il termine *debug* è un neologismo tecnico che non si trova sul dizionario. Poiché "bug" significa cimice, pensiamo che debug significhi qualcosa come "togliere le pulci" al programma.

Tutto il resto prosegue in un modo che è facilmente immaginabile (e comodamente sperimentabile da parte di chi possiede il sistemino). Punti salienti risultano quelli dopo l'esecuzione di PUSH HL (opcode E5 all'indirizzo D13): si ha SP = 0FFE, in quanto il puntatore allo stack è stato decrementato di due, nonché dopo CD 00 0D (ind. D14) con SP ancora decrementato di due (per la chiamata di MOLTIP) e PC = 0D00, indirizzo di tale routine. Dopo RET Z e POP HL si hanno invece i processi contrari. Sarebbero pure interessanti i dettagli del "giro" delle varie istruzioni di MOLTIP, ma la cosa, oltre che portarci via troppo spazio, a nostro parere risulta diseducativa: è infatti indispensabile che l'allievo conquisti con un certo sforzo personale la materia, senza continuamente esser preso per mano, come un fanciullo.

L'esecuzione passo passo, come si è visto, è interessante ma lenta e un po' noiosa. Spesso, quando si è abbastanza sicuri dal fatto proprio, è meglio tentare il collaudo diretto. Nel nostro caso si fa la battuta di "E" seguita da DOC (più, al solito, NL). Si osserverà l'accensione del LED di cui il NASCOM è dotato per segnalare l'avvenuta esecuzione dell'istruzione 76 (HALT). Se questo non avvenisse, ciò significherebbe che il programma sta eseguendo un loop senza termine o qualche altra diavoleria (dovuta *sempre* a un nostro errore!). Naturalmente non basta che il LED si accenda, occorre verificare che il risultato del prodotto sia giusto. In questo caso, poiché la CPU è in stato di halt, essa è sensibile solo ad un comando di reset (o un interrupt). Quindi battiamo il tasto RS (il "." NL qui non può operare) e con il comando M possiamo controllare il contenuto dei due byte E02 ed E03 in cui dovrà essere il valore corretto del prodotto.

Subroutine del NASBUG

Il monitor utilizza vari sottoprogrammi nel corso del suo funzionamento. Essi possono essere chiamati in gioco utilmente in un programma dell'utente, risparmiando aree di memoria (ed operando in modo abbastanza simile a quello che avviene nel software di un sistema, di media complessità, controllato con microprocessore). Qui citiamo solo quelle principali, tra cui quelle che avremo occasione di impiegare nei successivi esempi.

- KDEL all'indirizzo 0035 dà un ritardo di circa 5 - 6 millisecondi.
- KBD all'indirizzo 0069: fa la scansione della keyboard; se nessun tasto viene battuto, essa rientra con A = 0 ed il flag Cy = 0, altrimenti il carattere, in codice ASCII, corrispondente alla chiave premuta è caricato nell'Accumulatore e messo in "on" il Carry.
  È la routine che impiegheremo di più;

- CRT di indirizzo 013B, provvede a caricare sullo schermo, nella posizione attualmente occupata dal cursore, il carattere sito in A e a spostare di un posto a destra il cursore, facendo lo "scroll" se si supera l'ultima posizione di una riga dello schermo (°).

N.B. Se si impiega la CRT in un programma utente è buona norma posizionare l'indirizzo del cursore nel punto dello schermo da cui si vuol far partire la scrittura. Il monitor tiene l'indirizzo del cursore nelle posizioni di memoria 0C18 e 0C19.

Volendo ad es. partire dal punto sullo schermo in basso a sinistra (indirizzo 0B8A, v. Fig 3-4) si farà, solitamente all'inizio del programma:

Contenuto	Codice Assembly	Commenti
21 8A 0B	LD HL, 0B8AH	carica in HL indir. cursore
22 18 0C	LD (0C18H), HL	carica HL in cursore

Nel corso di un programma può anche essere utile, per andare a capo prima della fine riga, posizionare nuovamente il cursore in modo opportuno prima di chiamare CRT. La CALL CRT sarà poi solitamente preceduta da una istruzione tipo LD A, CAR (o LD A, n o simili) e CRT sarà stata inizialmente definita da una EQU:

```
CRT     EQU     0035H (o anche solo 35H)
```

Praticamente sempre negli esempi preferiremo tuttavia una tecnica di programmazione *diretta* dell'emissione video, e se ci siamo dilungati nella descrizione della routine CRT, oltre che per motivi di documentazione, l'abbiamo fatto per coloro che volessero sviluppare propri esempi, in cui impiegare tale subroutine.

La tecnica di controllo diretto di cui abbiamo parlato opera con le modalità seguenti:

- per scrivere un carattere - chiamiamolo genericamente CAR - in una determinata posizione, diciamo 0A0F: LD A, CAR seguito da LD (A0F), A;
- per scrivere una stringa di caratteri, in numero minore o uguale 48, sarà conveniente l'indirizzamento indiretto tramite HL, con l'uso di INC HL

(°) - Il NASBUG a tal fine utilizza caratteri 00H (caricati inizialmente col reset) posti nelle posizioni "invisibili": quando si invade una posizione con tale carattere si sa che la riga è finita.

per passare al carattere successivo, sulla destra, oppure impiegare istruzioni tipo LDIR (Load Increment and Repeat);
- per passare da una riga alla seguente (senza scroll, per il quale è senz'altro più conveniente la CRT; per contro, noi stiamo implicitamente riferendoci ad una scrittura dall'alto verso il basso), mantenendo la stessa colonna, aumentare di 40H il contenuto del puntatore HL. Ad es. (v. sempre Fig. 3-4) per passare da 09CA a 0A0A si farà:

(09CA + 40) H = 0A0AH

Il modo più pratico consiste nel tenere in DE il valore 0040H e utilizzare l'istruzione: ADD HL, DE.

Per tutti i problemi in cui si usa lo schermo sarà invece opportuno porre all'inizio del programma le istruzioni:

Contenuto	Codice Assembly
3E 1A	LD A, 1EH
CD 3B 01	CALL CRT

Il codice 1E è interpretato dal NASBUG (o meglio dalla CRT) come ordine di operare la pulizia dello schermo.

Infatti, dopo aver scritto tramite tastiera un programma, sullo schermo restano visualizzati i caratteri relativi a quei codici.

A rigor di termini il problema non si pone se si impiega, prima della esecuzione, il comando RS, che, come si è visto, inizialmente provvedde a questa occorrenza automaticamente. In generale è però preferibile il comando "." NL (che non fa saltare all'indirizzo zero e non provoca questo automatico blencaggio). Ciò perché, ad esempio, la cancellazione meccanica dello schermo non è sempre opportuna.

Modi di terminare un programma (sul NASCOM)

Un programma può essere *totalmente ciclico* (v. in seguito il programma per la generazione di un'onda quadra) e, se non è già previsto nel loop il "polling" (lett. *inchiesta*), ossia ad es. l'interrogazione della tastiera per una particolare lettera prevista per uscire dal ciclo (qualcosa del genere è illustrato nel diagramma di Fig. 1-3), l'unico modo possibile per interrompere il programma è tramite il comando RS. In un problema di controllo reale invece, come si è già detto, ciò è solitamente realizzato con un interrupt.

Il programma che invece prevede una fine può essere terminato, come si è già visto in fase di debug di P. 2-1, con l'istruzione HALT. Per semplicità porremo questo comando in altri programmini.

È però, oltre che più corretto, più comodo concludere con istruzioni per il ritorno al monitor. Così facendo, il NASCOM è già predisposto per la esecuzione di un altro programma esistente in memoria e/o la ri-esecuzione dello stesso, *senza necessità di battere RS.*
Queste istruzioni sono:

Contenuto	Codice Assembly	Commenti
31 33 0C	LD SP, 0C33H	inizializza lo stack del monitor
C3 59 03	JP 0359H	salta a STRTO

STRTO è un punto del monitor che si trova opportunamente *oltre* le istruzioni che puliscono lo schermo (di qui poi la necessità di provvedere all'inizializzazione dello stack a 0C33 che si trova pure nelle primissime posizioni). Se invece si fosse fatto semplicemente: JP 0 (in assoluto C3 00 00) surrogando a software il comando RS, si sarebbe avuta la cancellazione proprio dei risultati finali previsti dal programma.

La soluzione JP 0 va comunque bene in casi in cui non si usi lo schermo da parte del programma utente.

Riprendiamo ora la codifica degli altri programmini di cui si è visto il flow chart nel Cap. 1.

Programma P. 3-1. DIVISIONE PER RIPETUTE DIFFERENZE

Relativo al diagramma di Fig. 1-6, divisione per differenze ripetute. Limitiamoci per brevità alla sola subroutine, lasciando per esercizio al lettore il compito, che non dovrebbe essere troppo difficile, di codificare un piccolo "main" che partendo da dati in opportune celle di memoria (es. 0C50 e 0C51 per i dati DDO, DRE mentre 0C52 e 0C53 saranno destinate a contenere QUOT e REST) chiami poi la routine che stiamo per scrivere, dopo averli caricati nei registri: D ed E. Il registro invece che al termine conterrà il quoto è B, mentre il resto si troverà nell'Accumulatore A. Consigliamo di terminare il piccolo MAIN come suggerito poc'anzi (istruzioni di ritorno al monitor). La prova del programma potrà infine essere fatta ripetutamente, cioé caricando i dati diversi in 0C50 e 0C51, senza altro comando che M (anche per controllare i risultati).

La subroutine, che parte da 0C54, cioé dopo le aree dati, è codificata di seguito e, con i commenti di cui è corredata è di immediata comprensione, anche perché segue da vicino il flow. C'è solo una piccola variante relativa al deviatore SW.

Indirizzi	Contenuto	Label	Codice Assembly	Commenti
C54	AF	DIVIS	XOR A	A = 0 più reset del Cy
C55	BB		CP E	DRE = 0?
C56	C8		RETZ	Se SI rientra con Cy=0
C57	47		LD B, A	QUOT = 0
C58	7A		LD A, D	DDO in Acc. (poi REST)
C59	BB	PROVA	CP E	cfr. REST e DRE
C5A	D8		RET C	se REST<DRE rientra (con Cy = 1)
C5B	93		SUB E	REST − DRE in REST
C5C	04		INC B	QUOT + 1
C5D	18 FA		JR PROVA	
C5F		(segue MAIN di test)		

Si è qui trovato conveniente impiegare in tale funzione il flag Cy. Infatti nella comparazione, alla label PROVA, si verifica che, quando A<E (ossia REST minore DRE), *si esce con Cy = 1.* Poiché inoltre si opera con numeri positivi; per una volta abbiamo ritenuto poi di usare il test del Carry dopo la comparazione, facendo RET C, anziché RET M (che era del tutto equivalente). Tornando al nostro switch, poiché esso ora risulta settato nella condizione normale, bisognava *resettarlo* in quella anomala di DRE = 0. Ciò è automatico con la XOR in testa alla routine.

Il lettore faccia una prova anche per il caso DRE = 0, constatando, o con esecuzione passo passo o ponendo un break point *a valle* di CALL DIVIS, che si esce con divisione non eseguita e Cy = 0.

Programma P. 3-2. MASSIMO E MINIMO IN UN INSIEME

Codifica del flow di Fig. 1-8. Stavolta, concepiamo il lavoro come un programma a sé ed usiamo un po' a fondo l'Assembly. Riserviamo 16 posizioni di memoria per l'ipotetica tabella a partire da C50, utilizzando lo pseudo codice DEFS 10H, associato alla label INIZ, che pertanto è, come in Fig. 1-8, il valore iniziale del puntatore che qui, anziché k, sarà naturalmente la coppia solita HL. Seguiamo ora un po' più da vicino lo sviluppo del programmino (scritto più avanti). Dopo LD HL, INIZ, alla label EXEC, che, come si vede dal contenuto, è codificato con il caricamento in HL di C50, viene messo 16 in B, e (HL), cioé il primo elemento, in C e D, che sono i registri destinati a contenere MAX e MIN. Segue, alla label DECREM, l'istruzione DJNZ CONTIN. Questa decrementa B e, se diverso da zero, salta a CONTIN (salto relativo di + 8), dove il programma prosegue. Altrimenti si fa il ritorno al monitor. All'indirizzo C6F di CONTIN si incrementa HL, puntando ad altro elemento e caricandolo in A. Dopo il confronto con C, si salta a MAXIM con la condizione P (che corrisponde a maggiore o uguale,

onde il caricamento in C si fa anche in questo caso che a rigore sarebbe superfluo, ma si è preferito operare così per evitare ulteriori complicazioni, tanto il risultato non muta). Chez ... MAXIM si provvede a porre in C l'elemento puntato da HL, poi si torna, con salto relativo di −21 (in assoluto E9) a DECREM, cioé a togliere 1 da B. Se invece l'elemento è inferiore a MAX, non si salta e, all'istruzione di indirizzo C75 si fa la comparazione con D e, se minore, si va a MINIM. Si noterà che questa label è accanto ad una istruzione già utilizzata, LD D, (HL) in testa al programma. Qualora infine, neanche il salto ultimamente visto viene fatto, dato che il nuovo elemento non è nè superiore nè inferiore a MAX e, rispettivamente a MIN, all'indirizzo C79 si va semplicemente alla solita DECREM (salto relativo di −18, contenuto EC).

Programma P. 3-2

Indirizzi	Contenuto	Label	Codice Assembly	Commenti
			ORG C50H	
C50		INIZ	DEFS 10H	riserva 16 posizioni
C60	21 50 0C		LD HL, INIZ	inizializza puntatore
C63	06 10		LD B, 10H	B = 16
C65	4E		LD C, (HL)	Elemento in MAX
C66	56	MINIM	LD D, (HL)	Elemento in MIN
C67	10 06	DECREM	DJNZ CONTIN	B −1: se B=0 CONTINUA
C69	31 33 0C		LD SP, C33H	ritorno al
C6C	C3 59 03		JP STRTO	monitor
CF6	23	CONTIN	INC HL	punta ad altro elemento
C70	7E		LD A, (HL)	ponilo in Accumulatore
C71	B9		CP C	cfr. con MAX
C72	F2 7B 0C		JP P, MAXIM	se magg. o uguale, va MAXIM
C75	BA		CP D	altrim. cfr. con MIN
C76	FA 66 0		JP M, MINIM	e se minore salta a MINIM
C79	18 EC	JRDEC REM		altrim. torna a DECREM
C7B	4E	MAXIM	LD C, (HL)	elem. in MAX
C7C	18 E9		JR DECREM	e ritorna a DECREM.

Si va avanti finché non risulta B = 0.

Prova del programma. Consigliamo di operare come segue: caricare 16 numeri a caso (ci si può basare su quelli stessi che si trovano già in memoria) col comando M seguito da C50 e seguenti. Quindi, dopo aver inserito un break point all'indirizzo C67, col comando BC67, in corrispondenza della label DECREM, presso la quale si decrementa B finché non è nullo, operare l'Execute con il comando EC 60 (e non, come qualche distrattone farebbe, con EC50!). Verranno eseguite le istruzioni fino a C67 e si leggeranno, tra

l'altro, C = D = 1° elemento (che si avrà cura di aver segnato a parte, insieme agli altri 15). Per far ripartire il programma basterà battere E seguito da NL: l'esecuzione riprenderà, sempre arrestandosi a DECREM. Facendo 16 volte questa operazione, si potrà verificare che in C e D vanno successivamente il MAX e MIN *momentanei* fino al risultato definitivo.

Programma P. 3-3 (si riveda il flow di Fig. 1-9). SORT DI UN INSIEME

Rispetto al flow, i puntatori j e k sono ora le coppie di registri-puntatori DE ed HL, mentre il decontatore B è l'omonimo registro. Seguire ora passo passo come nel caso precedente porterebbe via troppo spazio. Così lasciamo la pena al paziente lettore. Facciamo invece alcune osservazioni di carattere formale: l'uso della EQU, con il valore 16 dato direttamente in decimale (idem per lo pseudo opcode DFS che riserva, come nel programma che precede, 16 celle a partire da C50); l'uso di IND + 1 nell'istruzione che segue evidenzia bene il fatto che HL punta al byte accanto (ind. C51) e l'analoga, all'indirizzo C66 che assegna a B il valore di N − 1, anch'essa abbastanza eloquente, se si ricorda l'algoritmo. Naturalmente, si poteva anche banalmente fare: LD B, 15 oppure LD B, N ove N era definito di valore 15 dalla EQU.

Programma P. 3-3

Indir.	Contenuto	Label	Codice Assembly	Commenti
		N	EQU 16	N=n° elementi
			ORG C50H	
C50		IND	DEFS 16	
C60	11 50 0C	DACAP	LD DE, IND	carica valori
C63	21 51 0C		LD HL, IND + 1	iniziali dei puntat.
C66	06 0F		LD B, N − 1	lungh. tab. meno 1
C68	1A	CONFR	LD A, (DE)	confr. un
C69	BE		CP (HL)	elemento col successivo:
C6A	28 0A		JR Z, DECR	(salto + 12): se uguali o
C6C	FA 76 0C		JP M, DECR	minore il 1° va a DECR
C6F	08		EX AF, AF'	inizio scambio (HL) e (DE)
C70	7E		LD A, (HL)	
C71	12		LD (DE), A	
C72	08		EX AF, AF'	
C73	77		LD (HL), A	termine scambio
C74	18 FA		JR DACAP	ricomincia (salto − 20)
C76	05	DECR	DEC B	B = B − 1
C77	C8		RET Z	se B = 0 rientra
C78	13		INC DE	prepara nuovi valori dei
C79	23		INC HL	puntatori DE e HL
C7A	18 EC		JR CONFR	rivà al cfr. nuova coppia di elementi (salto − 18)

L'unico punto che riteniamo di commentare da vicino si riferisce alle istruzioni per lo scambio di una coppia di elementi. La cosa più eloquente è di illustrare i vari passaggi con la tabella che segue.

Istruzione	Reg. A	Reg. A'	posiz. (DE)	posiz. (HL)
—	x	-	x	y
EX AF, AF'	-	x	x	y
LD A, (HL)	y	x	x	y
LD (DE), A	y	x	y	y
EX AF, AF'	x	y	y	y
LD (HL), A	x	y	y	x

Nella prima riga si ha la situazione prima dello scambio, nelle altre quella conseguente a ciascuna istruzione.

Si poteva anche usare il solo puntatore HL (°): nel punto centrale della routine si carica (HL) in A, si incrementa HL, si confronta A col nuovo valore puntato da HL ecc. Lasciamo il lettore a deliziarsi in questo piccolo rompicapo. Avvertiamo anche che l'algoritmo proposto del flow-chart in oggetto non è l'unico possibile, altri ve ne sono reperibili nella letteratura specializzata e codificabili in linguaggio Z80 (°°).

Programma P. 3-4 (flow chart di Fig. 3-5)

MOLTIPLICAZIONE TRA INTERI BIT PER BIT

A conclusione di questo capitolo, non possiamo esimerci dall'illustrare la reale routine con la quale si compie la moltiplicazione tra interi.

Infatti quella illustrata in Fig. 1-5 e nella codifica P. 2-1 al pregio della estrema semplicità unisce però la eccessiva lentezza (se il MRE vale 255 si fanno altrettanti loop).

Nella Fig. 3-5 illustriamo pertanto con un flow l'algoritmo per la moltiplicazione tra interi non segnati a 16 bit, la cui codifica è stata presa dal citato manuale Z80-CPU.

(°) - Si tenga presente che HL è un puntatore privilegiato rispetto a DE e BC: con esso sono pos sibili più operazioni che con gli altri.

(°°) - Nel manuale Z80 ce n'è uno fatto col criterio detto "delle bolle": è più veloce, ma più complicato e lungo del nostro (che presenta un piccolo dettaglio di originalità). D'altronde il nostro obiettivo è principalmente quello di proporre punti di discussione, in funzione dell'addestramento.

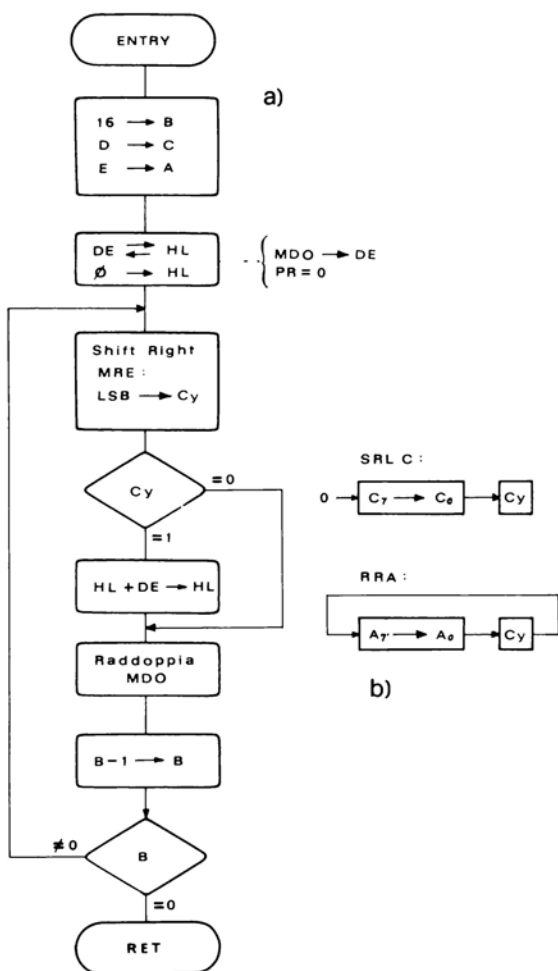



Figura 3-5 Prodotto tra interi, in doppia precisione, bit per bit.

Contenuto registri:

- Coppia HL, all'inizio MDO, al termine PR;
- Coppia DE = MRE, poi MDO;
- Reg. C = bit di ordine più alto di MRE;
- Reg. A = bit di ordine inferiore di MRE.

Il registro B funge da de-contatore e, caricato inizialmente con 16, serve a numerare i 16 passi dell'algoritmo, che poi è quello che viene eseguito nelle operazioni in binario carta e matita: estrarre da MRE un bit, dal più basso al

più alto, quindi aggiungere a PR, fatto inizialmente = 0, il MDO se il bit è 1 (altrimenti non si aggiunge nulla). Poi, prima di un nuovo ciclo, si moltiplica per due il MDO.

Commentiamo più da vicino il diagramma, che è stato orientato abbastanza nettamente al linguaggio: dopo il trasferimento di D in C ed E in A, con lo scambio DE con HL, ed il successivo HL = 0 il MDO va in DE e PR = 0. Viene poi effettuato lo shift destro di MRE, con il (Least Significant Bit) che da A_0 va in Cy.

Ciò si ottiene (v. oltre la codifica, all'indirizzo D08) con SRL C seguito da RRA. Come risulta in b), stessa figura, il primo shift porta il bit C_0 nel Carry (e 0 in C_7 ecc.), mentre RRA (rotazione a destra di A, insieme a Cy) porta Cy in A_7 e A_0 in Cy.

Così facendo, è come se l'insieme dei registri C ed A fosse un unicum , con i bit che scorrono da 1° al 2° registro e da questi a Cy. Si testa poi Cy e se è 1 si aggiunge DE ad HL, cioé MDO al PR, altrimenti si salta questa operazione. Quindi si provvede a moltiplicare per 2 MDO (istruzioni da D0E A D10, v. codifica): con EX DE, HL si pone MDO in HL, accantonando temporaneamente in DE il valore attuale di PR; poi si fa ADD HL, HL cioé aggiungi ad HL se stesso, di evidente significato, infine con la nuova EX DE, HL si ripristinano nelle coppie di registri soliti il nuovo MDO e PR. La routine termina con il decremento di B: quando esso vale 0 si ha RET, altrimenti si riprende con lo shift di MRE.

Programma P. 3-4

Indirizzi	Contenuto	Label	Codice Assembly	Commenti
D00	06 10	MOLT16	LD B, 16	n° bit da shitfare
D02	4A		LD C,D	trasferisci MRE
D03	7B		LD A, E	
D04	EB		EX DE, HL	MDO in DE
D05	21 00 00		LD HL, 0	PR = 0
D08	CB 39	GIRO	SRL C	shifta MRE, con
D0A	1F		RRA	LSB in Cy
D0B	30 01		JR NC, VIA	se non Cy va VIA
D0D	19		ADD HL, DE	se no agg. MDO a PR parziale
D0E	EB	VIA	EX DE, HL	MD0 in HL
D0F	29		ADD AL, HL	molt. HL per 2
D10	EB		EX DE, HL	ripristina situazione
D11	10 F5		DJNZ GIRO	ripeti finché vi sono bit
D13	C9		RET	

"Nella sua follia c'è del metodo" (Amleto, atto II)

CAPITOLO 4
GIOCANDO COL MICROCOMPUTER

Come abbiamo detto nell'Introduzione, nella programmazione occorrono anche doti di fantasia . Molti programmatori anzi sono, almeno in apparenza, delle teste matte. Senza arrivare a certi eccessi (del tipo: far disegnare donnine più o meno svestite all'elaboratore) e rimanendo nell'ambito di programmi di complessità, al più, media vogliamo proporre degli esempi che da un lato si presentino abbastanza divertenti e tali da stimolare negli allievi l'interesse (ma chi l'ha detto che studiando ci si debba sempre e solo annoiare?); d'altro canto spesso si dimostreranno più o meno indirettamente imparentati con applicazioni reali.

A tale scopo risultano molto utili la tastiera del NASCOM e la routine KBD, nonché l'unità video. Questa costituirà, vedremo, anche un mezzo alternativo di "debugging" anche per programmi applicativi, con la comparsa al momento giusto di caratteri e/o messaggi, del tutto equivalenti ad altro tipo di emissione di segnali in sistemi di controllo reali.

Prima di procedere, vediamo alcuni metodi mediante i quali ottenere dei *ritardi* temporali più o meno lunghi tramite software.

Ritardi software

Nelle applicazioni di ogni tipo, ed in primis in quelle industriali, essi sono di primaria importanza. Il caso più evidente è quello della temporizzazione di un processo: il lettore non troppo addentro nelle cose tecniche pensi ad una lavatrice domestica, in cui la durata di ogni fase (lavaggio, risciacquo ecc.) è abbastanza precisa. Se vogliamo comandare una simile macchina con un microprocessore, la cosa migliore è generare questi ritardi con programmini in cui si eseguono a vuoto delle istruzioni che, di per sé, non servono a nulla. Tenendo conto che il microprocessore opera sulla base di un clock quarzato e che ogni istruzione ha, come si è visto, una durata corrispondente ad un ben preciso numero di periodi (i cosiddetti *stati-T*), basterà inserire in un loop queste istruzioni "inoperose" utilizzando un decontatore per farle eseguire un numero adeguato di volte, per ottenere il tempo che si vuole.

Per concretizzare il discorso, vediamo come è fatta la routine KDEL del NASBUG:

Indirizzi	Contenuto	Codice Assembly			
0035	AF	KDEL	XOR	A	
36	F5	SU	PUSH	AF	
36	F1		POP	AF	
37	F5		PUSH	AF	
38	F1		POP	AF	
39	3D		DEC	A	
3A	20 F9		JR		NZ SU
3C	C9		RET		

Le istruzioni oziose nel loop sono il succedersi, per due volte, di un PUSH seguito da POP, che, evidentemente, lascia le cose come erano. Però, così facendo, trascorrono un numero non esiguo di microsecondi, in quanto si tratta di istruzioni singolarmente abbastanza lunghe. Come si può vedere dall'Appendice B, i periodi T sono, per PUSH, 11 e, per POP, 10, ossia un totale di: $(11 + 10) \times 2 = 42$ stati T. Poiché inizialmente si fa $A = 0$, il loop è percorso *256 volte* in quanto la prima volta che si esegue DEC A, si passa a FF e, quindi, restano *altri 255 cicli da fare dopo il primo*. Ciò fa complessivamente un numero di T-state pari a $256 \times 42 = 10.752$. Assumiamo una frequenza del clock tra i 2 e i 2,5 MHz in modo che il ritardo dato da KDEL risulti (come poi si verifica anche sperimentalmente) di *circa 5 millisecondi.*

Per ritardi più brevi, si può o utilizzare una routine consimile (°) oppure caricare inizialmente nel decontatore un valore diverso da zero.

È quanto si può fare utilizzando CALL KDEL + 1 preceduta da un caricamento in A del valore opportuno.

Esempio: con LD A, 128 prima di CALL KDEL + 1 (cioé in assoluto CD 36 00) si ottiene un ritardo metà, circa 2,5 msec.

Ritardi medi e lunghi

Per ottenere ritardi maggiori (secondi o più) il criterio di massima è quello di inserire il loop più piccolo in uno più grande ecc.

Proponiamo una routine abbastanza compatta, che usa l'istruzione DEC BC.

(°) - Tanto per fare un esempio: caricare inizialmente, in A, 10000000B, poi eseguire un loop che comprende RRCA; si esce dal loop con Cy = 1 dopo 8 passi. Proposto al lettore come esercizio.

Il loop di ritardo è percorso un numero di volte pari al valore posto inizialmente in BC (il ciclo ha termine con BC = 0). Con valore iniziale BC = 0 si hanno 64 K = 65.536 "giri".

Possiamo ad esempio inserire in tale loop una routine tipo KDEL e si ha:

```
RITARD  CALL  KDEL
        DEC   BC
        XOR   A
        CP    B
        JR    NZ, RITARD
        CP    C
        JR    NZ, RITARD
        RET
```

Le istruzioni dopo DEC BC sono necessarie in quanto quest'ultimo *non* influenza il flag Z.

Fatto A = 0, si confronta prima con B, poi con C e solo se entrambi sono zero, si esce.

Un tale sottoprogramma va preceduto da un LD BC, val. Con val. = 0 il ritardo sarebbe pari a 65.356 × 5 = 326.000 msec, cioé 5,4 minuti.

Un'altra tecnica consiste nell'uso di istruzioni PUSH e POP.
Poniamo che PAUSA sia una routine che dà un certo ritardo (es. i circa 5 secondi trovati sopra). Si può fare:

```
BIGRIT  PUSH  BC
        CALL  PAUSA
        POP   BC
        DEC   BC
        XOR   A
        CP    B
        JR    NZ, BIGRIT
        CP    C
        JR    NZ, BIGRIT
        RET
```

Col PUSH di BC il vecchio valore di esso è salvato sullo stack, poi si esegue PAUSA che, supponiamo, utilizza internamente DEC BC ecc. come nel caso precedente e con valore iniziale di BC *fisso* (esempio: BC = 2500).

All'uscita da PAUSA il POP BC ristabilisce il precedente valore salvato, anche se PAUSA ha reso BC = 0. Segue DEC BC ecc. In definitiva il ritardo

PAUSA è eseguito un numero di volte pari a quello posto in BC prima della chiamata di BIGRIT.

Riteniamo che gli esempi che abbiamo fatto siano più che sufficienti.

Prima di proseguire, proponiamo a chi legge un quesito. Nella parte terminale dei due ultimi programmi, da XOR a RET esclusa, non si potevano mettere le istruzioni seguenti, risparmiando un byte?

```
LD   A, B
JR   NZ, BIGRIT
CP   C
JR   NZ, BIGRIT
```

La risposta è no, in quanto la istruzione di caricamento *non attiva i flag*.
Si può comunque fare, in alternativa:

```
LD   A, B
OR   A              per attivare Z in conformità al valore caricato
JR   NZ, BIGRIT
OR   C              valida come la CP C
JR   NZ, BIGRIT
```

Si noti la OR C che qui equivale a CP C, in quanto, con A = 0, solo se anche C è tale l'operazione 0R dà tutti bit zero e il flag Z è attivato.

Vediamo ora un primo programmino-gioco basato su ritardi. Lo chiameremo *"prova di riflessi"* in quanto la macchina "punisce" se non si batte un tasto entro un tempo prestabilito. Il diagramma a blocchi è in Fig. 4-1.

Programma P. 4-1 PROVA DI RIFLESSI

RIT1 e RIT2 sono i valori caricati nelle posizioni di memoria 0C50 e 0C51, il primo, mentre il secondo è in 0C52. Servono a fornire valori di ritardo variabili a piacere in fase di caricamento con il comando M.

La prima parte del diagramma serve, come è di immediata evidenza, a dare un ritardo pari a RIT1 volte i 5 msec. di KDEL. Quando BC = 0, viene fatto comparire il carattere "?" sullo schermo (in posizione, ad es., 0BCF).

Questo è il segnale di significato: "avete tot msec. per battere un tasto" (qualsiasi). Segue il caricamento di RIT2 in B, seguito da CALL KBD.

Si tenga ora presente che, per motivi di *"debouncing"* cioé per creare un *effetto antirimbalzo*, questa routine incorpora ritardi (solo dopo un tempo

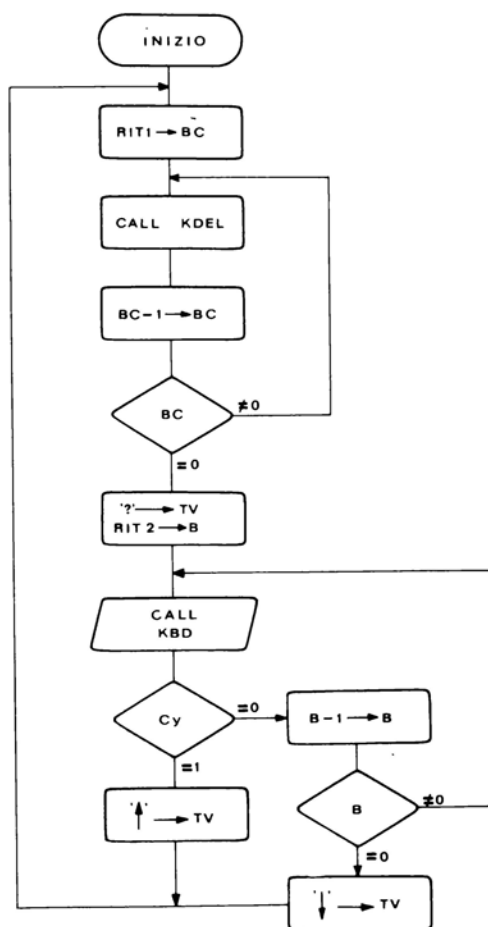



Figura 4-1 Prova di riflessi (time out).

necessario per l'assestamento dei rimbalzi del tasto il carattere è accettato come valido).

Sperimentalmente si trova che, quando nessun carattere è battuto si esce dalla KBD dopo cica 1,5 msec. Aggiungendo subito dopo la CALL KDEL si ha così un ritardo totale di 5 + 1,5 = 6,5 msec. Subito dopo si interroga lo stato del flag Cy, per quanto visto al Cap. 3, se Cy = 1 vuol dire che un tasto è stato premuto e in tal caso vien fatto comparire il carattere "freccia in su" sullo schermo, ad indicare battuta *entro* il tempo stabilito. Se invece Cy = 0,

si decrementa B e, con B = 0, si ritorna a CALL KBD, per offrire altri altri 6,5 msec. di tempo ecc. finché, con B = 0, si ha la condizione di *tempo scaduto* e la freccia in giù compare sul video, infamante simbolo di sconfitta (un po' come il pollice verso).

In entrambi i casi si ricomincia da capo, per un'altra prova.

I caratteri freccia sono codificati, nella ROM generatrice che è parte integrante del sistema display, con l'esadecimale 5E e 0B, rispettivamente.

Segue la codifica.

Programma P. 4-1

Indirizzi	Contenuto	Label	Codice Assembly	Commenti
		KDEL	EQU 35H	assegna i nomi KDEL TV e
		TV	EQU 0BCFH	
		KBD	EQU 69H	KBD agli ind. 35H, 0 BCF e 69H
			ORG C50H	
C50	10 27	RIT1	DEFW 10000	assegna val. 10000 a voce C50
C52	32	RIT2	DEFB 50	assegna valore 50 a byte C52
C53	ED 4B 50 0C	INIZIO	LD BC, (RIT1)	
C57	CD 35 00	SU	CALL KDEL	rit.do di circa 5 msec.
C5A	0B		DEC BC	
C5B	AF		XOR A	
C5C	B8		CP B	
C5D	20 F8		JR NZ, SU	salto rel. di − 6
C5F	B9		CP C	
C60	20 F5		JR NZ, SU	salto rel. di − 9
C62	21CF 0B		LD HL, TV	indir. 0BCF nel puntatore
C65	36 3F		LD (HL), 3FH	"?" trasf.to in TV
C67	3A 52 0C		LD A, (RIT 2)	carica in registro
C6A	47		LD B,A	B ritardo 2
C6B	CD 35 00	KEYB	CALL KDEL	(rit.do di circa 5 msec.)
C6E	CD 69 00		CALL KBD	attendi altri 1,5 msec.
C71	30 05		JR NC,GIU	se tasto non prem., vai GIU
C73	36 5E		LD (HL), 5EH	freccia in su, in TV
C75	C3 53 0C		JP INIZIO	nuovo gioco
C78	10 F1	GIU	DJNZ KEYB	c'è ancora (poco!) tempo ...
C7A	36 0B		LD (HL), 0BH	frecc. giù: tempo scaduto!
C7C	C3 53 0C		JP INIZIO	provi ancora, forse riesce.

Commentiamo le cose essenziali, perché con le scritte laterali più il già illustrato flow tutto dovrebbe essere self evident.

- *indirizzo C50:* il valore 10000 è tradotto dall'assemblatore in 2710H e caricato nella voce C50 "a rovescio", cioé, secondo le solite convenzioni dello Z80, low order 10 prima dell'high order 27;
- *indirizzo C53 e C67:* si notino le label tra parentesi, mentre sul flow-chart si è preferito scrivere senza le parentesi RIT1 RIT2, e intendendo indicare qui i valori anziché gli indirizzi in cui sono contenuti;

- *indirizzo C62:* il puntatore di memoria HL è posizionato "una tantum"; in tal modo, più avanti (istruz. C65, C70 e C77), si pongono i vari caratteri sempre in 0BCF.

Modalità esecutive: 1) caricare con il comando M i codici assoluti dalle posizioni 0C50 in poi, quindi, dopo il "." NL o RS attuare il comando E seguito da C53 (indirizzo di INIZIO): appena compare, sulla "top row", il punto interrogativo schiacciare al più presto un tasto qualsiasi, eccetera.

Per variare RIT2 (o anche RIT1, allo scopo ad es. di porre un più lungo intervallo tra una prova e l'altra) non c'è altro da fare che RS, seguito M C50 + nuovo valore.

Il lettore può anche divertirsi a escogitare delle varianti.

Proponiamo le seguenti: inserire all'inizio le istruzioni LD A, 1EH seguita da CALL CRT per avere la pulitura schermo dopo il caricamento del programma, anche senza fare RS (o schiacciare insieme i due tasti shift e back-space); attenzione però che ora JP INIZIO non va più bene, se fa saltare a tali istruzioni preliminari (basterà introdurre un'altra label, es. ANCORA, dopo di esse), perché ne seguirebbe l'immediata cancellazione delle frecce che indicano il risultato della prova. Altre varianti sostituire "?" e le frecce con messaggi più o meno spiritosi; sostituire l'attesa iniziale con un CALL KBD seguito da JR NC, 3 che attende un carattere dalla tastiera: il punto interrogativo compare solo dopo che, mettiamo, un primo giocatore ha battuto un tasto; infine, un po' più impegnativo, prevedere l'indicazione del tempo impiegato per dare la risposta. (°).

Il giochetto che abbiamo visto, facciamo rilevare, non è del tutto frivolo. Infatti esso viene incontrato, mutatis mutandis, in quelle applicazioni industriali in cui occorre verificare che un evento arrivi o meno entro un termine prestabilito (problemi cosidetti di *"time out"*).

Programma P. 4-2

TABELLA ORDINATA ALL'ATTO DEL CARICAMENTO

Passiamo ora ad un problemino più serioso. Vogliamo caricare una tabella introducendo i dati da tastiera, ma (ecco il difficile) man mano che si ha un nuovo dato esso deve occupare il posto che gli compete in modo che la tabella sia sempre ordinata in modo crescente.

Descriviamo l'algoritmo a parole (flow chart lasciato per esercizio a chi legge). Ogni volta che arriva un nuovo dato, esso viene confrontato successivamente col primo, secondo ecc. elemento. Inizialmente, la tabella è riem-

---

(°) - Le istruzioni da inserire dopo Cy = 1 sono: LD A, (RIT2) seguita da SUB B, più display di tale risultato su schermo TV. Si otterrà un valore da moltiplicare per (circa) 6,5 msec.

pita con tutti blank. Se allora in una certa posizione cui si punta vi è un blank, senz'altro si carica il dato nuovo, altrimenti si opera il confronto tra il nuovo dato e quello della posizione puntata: se è maggiore si passa avanti, altrimenti si spostano a destra tutti i dati della tabella e, liberatosi il posto che compete al nuovo dato, lo si carica lì.

Se il nuovo dato è maggiore di quelli finora caricati, si va sempre avanti finché non si trova una posizione a blank, dove lo si carica.

Poiché siamo in fase di divertissement, utilizziamo, come tabella, la "top row" (indirizzi da 0BCA e 0BF9 della memoria video). Inoltre, teniamo presente che: 1) i dati introdotti in tastiera tramite la routine KBD sono in ASCII, e analogamente la rappresentazione visualizzata sullo schermo è anch'essa corrispondente, per le cifre decimali e le lettere, a tale codice; pertanto, per semplicità, qui ci limiteremo a dati decimali, però di *due cifre*; 2) dato che lo scopo è di stupire gli amici, interporremo, tra una coppia e l'altra, uno spazio blank. In conclusione quindi occorrono tre byte della riga per ogni coppia di cifre, il che fa un totale di 48 : 3 = 16 numeri a due cifre scrivibili in tutto. Ricordiamo anche che gli indirizzi estremi della top row sono: 0BCA e 0BF9 (si riveda Fig. 3-4).

A questo punto, facciamoci coraggio e passiamo alla codifica (v. pag. 73).

Definite le label INTAB, FINTAB e BLANK, rispettivamente per gli indirizzi 0BCB (in 0BCA *partiamo con un blank),* 0BF9 e per il codice ASCII del blank, nelle successive istruzioni si caricano da tastiera prima una cifra decimale, poi l'altra. Si noti la CALL KBD seguita da un salto relativo di − 3 sulla condizione NC: come si è detto la KBD rientra nel MAIN con Cy = 1 se e solo se un tasto è stato premuto; con tale salto, si ritorna a KBD, attendendo ancora. Le due cifre introdotte sono poste in B e C.

Si posiziona poi HL a INTAB, si confronta la cella (HL) con il blank e, se uguale, con le istruzioni della label CARICA a NOBLK esclusa si pongono in (HL), poi in (HL + 1) le cifre in B e in C, poi si ritorna a KEYB1, per attendere altre battute. Quando in (HL) non v'è il blank, si salta a NOBLK: qui B è comparata con (HL) e, sulla condizione M (Minus) si va senz'altro a SPOST, dove, vedremo, si provvede ad eseguire lo scorrimento in avanti della tabella, come già detto. Questo perché, se la prima delle due nuove cifre è minore della corrispondente in (HL), non c'è bisogno di confrontare anche l'altra per sapere che il nuovo numero *precede* quello presente in tabella.

Se M non è verificata, con INC HL si passa al byte accanto e, con Z = 1, si salta ad ALTCIF, perché, pur essendo le prime due cifre uguali, occorre ancora confrontare le altre due.

Programma P. 4-2

Indir.	Contenuto	Label	Codice Assembly	Commenti
		INTAB	EQU 0BCB	indir. inizio e
		FINTAB	EQU 0BF9	indiri. fine tabella
		BLANK	EQU 20H	
C7C	CD 69 00	KEYB1	CALL KBD	attendi di un carattere
C7F	30 FB		JR NC, KEYB1	salto di − 3
C81	47		LD B,A	prima cifra in B
C82	CD 69 00	KEYB2	CALL KBD	altro carattere
C85	30 FB		JR NC, KEYB2	
C87	4F		LD C, A	seconda cifra in C
C88	21 CB 0B		LD HL, INTAB	
C8B	3E 20	ANCOR	LD A, BLANK	
C8D	BE		CP (HL)	(HL) è blank?
C8E	20 05		JR NZ, NOBLK	se NO vai a NOBLK
C90	70	CARICA	LD (HL), B	prima cifra in (HL)
C91	23		INC HL	passa accanto
C92	71		LD (HL), C	ponici seconda cifra
C93	18 E7		JR KEYB1	va ad attendere altro numero
C95	78	NOBLK	LD A, B	se non blank, con-
C96	96		SUB (HL)	fronta cifra con (HL)
C97	FA AA 0C		JP M, SPOST	se minore vai a SPOST
C9A	23		INC HL	passa a posiz. accanto
C9B	28 04		JR Z, ALTCIF	se = va a confr. 2ª cifra
C9D	23	INCR	INC HL	passa alla
C9E	23		INC HL	coppia accanto
C9F	18 EA		JR ANCOR	per fare altro confronto
CA1	7E	ALTCIF	LD A, (HL)	
CA2	91		SUB C	confr. 2ª cifra (°)
CA3	28 F8		JR Z, INCR	se ancora uguale, coppia accanto
CA5	30 02		JR NC, AV	se NC, (HL)>C: va ad AV
CA7	18 F4		JR INCR	altrim. va a INCR
CA9	2B	AV	DEC HL	aggiusta HL
CAA	C5	SPOST	PUSH BC	salva BC e
CAB	3E F7		LD A, F7H	calcola la
CAD	95		SUB L	lunghezza
CAE	4F		LD C, A	del blocco
CAF	06 00		LD B,0	da spostare verso destra
CB1	E5		PUSH HL	salva HL
CB2	11 F9 0B		LD DE, FINTAB	ultima posiz. in DE
CB5	21 F6 0B		LD HL, FINTAB − 3	quart'ultima in HL
CB8	ED B8		LDDR	sposta blocco
CBA	E1		POP HL	ripristina HL
CBB	C1		POP BC	e BC
CBC	C3 90 0C		JP CARICA	ora puoi caricare B e C.

Nota: (°) - andava bene anche CP C.

Osservazione: INC HL che precede la JR NZ, ALTCIF *non influenza* il flag Z (v. Appendice B), sicché questo è ancora corrispondente dall'esito della precedente comparazione. Ciò, come è chiaro, qui fa abbastanza comodo, altrimenti si sarebbe dovuto mettere INC HL di posizione C9A dopo JR Z, ALTCIF e far precedere la prima istruzione di ALTCIF da un altro INC HL.

A questo punto, istruzione C9D, ci troviamo con la cifra di B *maggiore* di quella in (HL), non c'è perciò bisogno di confrontare la cifra in C e, con due INC HL, si passa ad ANCOR per ripetere i confronti con la coppia accanto, e così via, finché eventualmnte non vi si trovi un blank ecc.

Torniamo ora ad ALTCIF. Qui si confronta la seconda cifra con (HL): se uguale si va a INCR e di lì si passa alla coppia accanto ecc. in quanto è inutile spostare alcunché, se entrambe le cifre sono uguali alle corrispondenti di tabella (non è bello scomodare chi ha gli stessi numeri ma ... precede per anzianità). Si testa poi il Carry: come si è detto, la condizione NC equivale a: 1° termine maggiore del 2°. Con NC si va ad AV, altrimenti, essendo la cifra in (HL) minore di quella in C, a maggior ragione che nel caso appena visto, si va a INCR ecc. Su AV si aggiusta HL, prima di SPOST.

Spieghiamo perché, in AV, c'è il decremento di HL. È semplice: noi proveniamo dal confronto delle *seconde* cifre, perciò ci troviamo un byte *a destra* rispetto al caso in cui siamo saltati direttamente a SPOST dal confroto delle prime. Con DEC HL ci si riporta in entrambi i casi nelle medesime condizioni.

Vediamo infine come opera lo scorrimento di cui abbiamo parlato. Si salva la coppia di cifre in BC, poi si calcola la lunghezza del blocco: F7 è il secondo byte dell'indirizzo delle ultime tre posizioni della "top row", quelle cioè, blank compreso, dell'ultima coppia di tabella; gli si toglie L, cioè il secondo byte del valore *attuale* del puntatore (N.B. i primi byte qui sono sempre 0B, perciò la loro differenza è sempre zero), si pone il risultato in C ed in

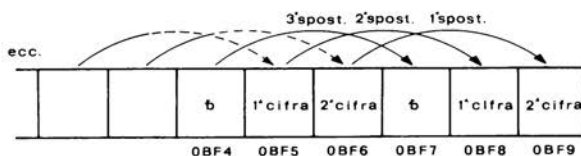



Figura 4-2

B zero. Così in BC c'è la lunghezza del blocco. Dopo che si è posto in HL l'indirizzo della quart'ultima cella ed in DE quello dell'ultima, basta invocare la potente LDDR perchè lo Z80 metta: quart'ultima posizione in ultima poi, dopo decremento di HL e DE, quint'ultima in penultima ecc. Come risulta chiaro dalla Fig. 4-2, ciò ha l'effetto di far scorrere ogni gruppo di tre byte nei tre byte accanto, ciò fino a che non si ha BC = 0.

A questo punto, non c'è che da ripristinare HL e BC (attenzione all'ordine: POP BC prima di POP HL, si è già detto ma insistiamo, *è errato*!) e saltare a CARICA per porre nelle posizioni lasciate libere la nuova coppia BC nel posto che le compete, lasciato libero grazie a SPOST.

Per concludere, facciamo le seguenti osservazioni:

- si è iniziato questo programma all'indirizzo C7C ossia *di seguito* al programma precedente, che terminava in C7B; ciò allo scopo di avere questo ed altri programmini consimili tutti caricabili/scaricabili su/da cassetta *unica*, senza problemi; come conseguenza formale la ORG non è qui più richiesta, dato che l'Assemblatore vede il tutto come un "unicum; così pure non occorre ridefinire KBD.
- c'è qui una imperfezione: se si vogliono caricare più di 16 numeri - cosa sia pure erronea, in quanto la tabella è prevista di 16 posti - si hanno degli inconvenienti:
  Se un numero è minore di uno dei già presenti, viene "distrutto" l'ultimo (cancellato dallo spostamento del penultimo elemento nell'ultimo); se invece un numero è maggiore di tutti i 16 presenti, HL punta a posizioni di memoria fuori della tabella, il che è privo di senso e dà risultati "buffi" sul video del NASCOM.

Il programma visto è una alternativa all'altro, più semplice, consistente nell'aggiungere il nuovo elemento *in coda* alla tabella, facendone poi il "sort" (routine già vista al capitolo precedente). È questo un esercizio proposto al lettore adattando quel 'sort' al caso attuale e/o riferendosi al caso "più umano" di una tabella normale di elementi a un sol byte.

Questa variazione può essere realizzata anche con il programma P. 4-2.

Programma P. 4-3.

ONDA QUADRA CON DUTY CYCLE VARIABILE

Si vuol far comparire ora sullo schermo un disegno fatto di asterischi (va pure bene qualsiasi altro carattere, come il '.', il '-' ecc.) come illustrato in basso nella Fig. 4-3.

L'asterisco parte dalla prima posizione, 0A4A, della riga omonima, quindi si salta rapidissimamente alla 0A0A, poi, dopo un ritardo RIT, l'asterisco viaggia alla posizione accanto ecc. Dopo n1 volte, esso salta alla riga inferiore, in tempo praticamente nullo, proseguendo, un RIT alla volta -

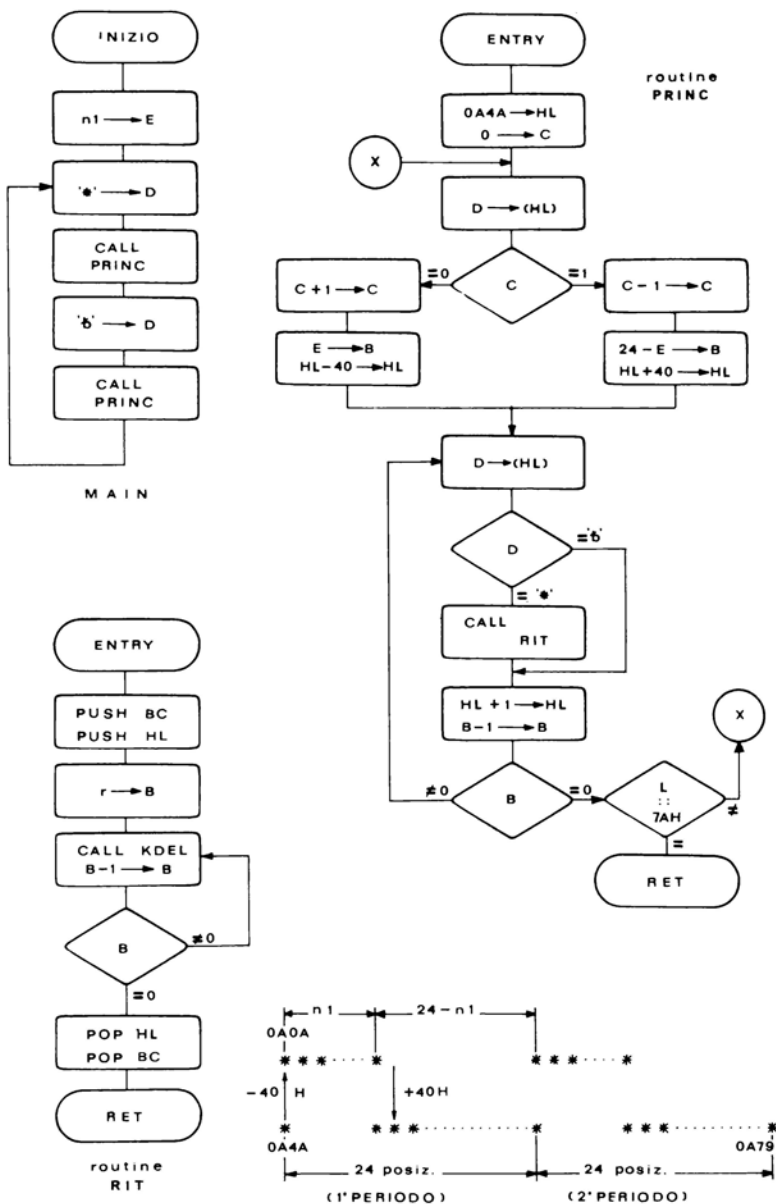



Figura 4-3 Onda quadra con "duty cycle" variabile.

per altri 24 - n1 posti, poi salta "in su", viaggia per n1 celle, ritorna "in giù" e, dopo altre 24 - N1 arriva in fondo alla riga inferiore (indirizzo 0A7A). In sostanza, si disegnano sulle due righe due alternanze complete. Il "*duty cycle*" è dato dal rapporto di n1 con 24, che visualizza il periodo completo.

Subito dopo questo disegno, esso viene cancellato facendo ripercorrere al puntatore HL la stessa successione di valori, ma *con il blank al posto dell'asterisco e senza il ritardo* RIT tra una posizione e quella accanto. Si simula così un "blanking" instantaneo. Poi si ricomincia con l'asterisco.

Si usano i registri e byte seguenti:

- *indirizzo 0CBF:* contiene il dato n1 (anche qui, si parte con la posizione successiva all'ultima del programma P. 4-2 precedente);
- *indirizzo 0CC0:* serve al dato, chiamiamolo *r*, pari al numero di volte i 5 msec. della KDEL corrispondenti al ritardo dato dalla routine RIT;
- *registro B:* decontatore sia in RIT che, vedremo tra breve, nella routine PRINC;
- *registro C:* utilizzato, ex abundantia, come switch: col valore C = 0 indica la necessità di saltare "in su", con C = 1 quella di saltare "in giù" tra le due righe;
- *registro D:* contiene, alternativamente, '*' o blank (codici, rispettivamente 2AH e 20H);
- *registro E*: immagazzina n1 inizialmente (onde in seguito ne sia più agevole il richiamo).

Vediamo ora il diagramma a blocchi di Fig. 4-3.

Si tratta di un esempio, crediamo, non banale di applicazione di routine. La prima, RIT, non richiede altri particolari commenti che le istruzioni PUSH BC e PUSH HL in testa ad essa e le corrispondenti POP prima del RET. Questa è una situazione che si può presentare spesso in sottoprogrammi che utilizzino registri che servono anche al programma chiamante (°). La RIT dà, come si è detto, un ritardo di r volte 5 msec.

(Nella codifica, si è scelto r = 80H, cioè 128, il che dà un tempo di un pò più di 0,5 sec tra un asterisco e l'altro, con una percorrenza piuttosto lenta del carattere).

Il programma MAIN è molto semplice: dopo il caricamento in E di n1, si pone in D '*' e si chiama PRINC, poi in D si mette il blank, si richiama PRINC poi si ricomincia con l'asterisco: è un loop senza fine interrompibile solo da RS.

(°) - Andava pure bene (anzi, meglio), la EXX, in testa e in coda.

Vediamo allora come opera questo PRINC tanto invocato.

All'inizio in HL va 0A0AH, prima posizione della riga omonima e in C zero.

Viene quindi posto D in (HL), *cioè asterisco o blank a seconda dei casi.* Lo switch C è poi interrogato. Inizialmente esso è zero, perciò subito dopo (blocco a sinistra) lo si pone a 1 (con la istruzione INC C, più "economica" che non LD C, 1 o anche del set di un qualche bit C_7, C_6....). Il contenuto di E, ossia n1, va poi in B, *mentre HL è diminuito di 40H.* Questo ha l'effetto di spostare il puntatore da 0A4A a 0A0A (ossia, come si è detto in generale, saltare da una riga alla precedente). D viene allora posto in (HL): si noti che, fino ad ora, la RIT non è intervenuta onde i due asterischi si vedono comparire simultaneamente, *a simulare tempo di salita nullo.* Si va quindi ad interrogare il registro D.

Anche questo ha un funzione di *deviatore*: con D = '*', si opera la chiamata di RIT, mentre con D = blank la si bypassa. L'uso di questo deviatore è abbastanza interessante. Se non avessimo usato questo piccolo trucco, avremmo dovuto escogitare due routine diverse tra loro solo sul particolare che l'una incorpora e l'altra no la RIT.

Nel blocco seguente, con INC HL si passa alla posizione seguente, si decrementa B ecc. in modo da continuare il caricamento di asterischi, o blank, nelle prime n1 posizioni della riga che inizia a 0A0A. Quando B =0, si interroga L e, se diverso da 7AH, si ricomincia da: D in (HL), tramite il connettore X. Al secondo passaggio però C = 1 (e, dopo aver rimesso a zero, con DEC C (1 − 1 = 0!), tale registro, si mette in B il valore 24 − n1, che corrisponde al numero di asterischi (o blank) da scrivere nella seconda parte del periodo, mentre HL è aumentato di 40H, per passare di nuovo alla riga in basso, quindi si riprende come prima ecc.

A questo punto il lettore non ha più alcuna difficoltà a seguire, passo passo, gli altri due giri della routine, in cui si completa il secondo periodo dell'onda quadra, finchè si arriva a L = 7AH, cioè al termine della riga.

Quanto alla codifica, essa non dovrebbe presentare particolari difficoltà, anche se i commenti sono un pò scarsi e vi sono alcune lievi rozzezze. Il lettore può ad esempio perfezionare, utilizzando la più compatta istruzione di aritmetica a 16 bit: SBC HL, DE (con DE = 40H), in luogo delle istruzioni da CDD a CE4, consistenti nel fare prima L − 40H, quindi nel sottrarre (sempre tramite l'Accumulatore) da H *zero con carry* (qui: prestito, che può avvenire sia richiesto all'ordine maggiore H. Analogo discorso vale per le istruzioni da CEC a CF3 (utilizzare ADD HL,DE) (°).

(°) - Si ricordi di fare il reset di Cy, prima di SBC e di salvare il salvabile.

Programma P. 4-3

Indirizzi	Contenuto	Label	Codice Assembly	Commenti
CBF	08	N1	DEFB 8H	va bene ogni val. tra 1 e
CC0	80	R	DEFB 80H	17 H (23 D)
CC1	21 BF 0C	ONDAQUAD	LD, HL, N1	inizio MAIN
CC4	5E		LD E, (HL)	
CC5	16 2A	A1	LD D, 2AH	asterisco in D
CC7	CD D1 0C		CALL PRINC	
CCA	16 20		LD D, 20H	blank in D
CCC	CD D1 0C		CALL PRINC	
CCF	18 F4		JR A1	Va ad A1 (salto di −10)
CD1	21 4A 0A	PRINC	LD HL, 0A4AH	inizio PRINC
CD4	0E 00		LD C,0	switch off
CD6	72	X	LD (HL),D	
CD7	AF		XOR A	A = 0
CD8	B9		CP C	cfr. C con zero
CD9	20 0C		JR NZ,A2	se NZ, va ad A2
CDB	0C		INC C	switch on
CDC	43		LD B,E	
CDD	7D		LD A,L	sottrai da L
CDE	D6 40		SUB 40H	40H (64 decimale)
CE0	6F		LD L,A	
CE1	7C		LD A,H	e da H zero, con
CE2	DE 00		SBC 0	riporto, per fare HL - 40H
CE4	67		LD H,A	(riga sopra)
CE5	18 0D		JR A3	va ad A3
CE7	0D	A2	DEC C	switch off
CE8	3E 18		LD A,24	fai 24 meno
CEA	93		SUB E	E in
CEB	47		LD B,A	registro B
CEC	7D		LD A,L	
CED	C6 40		ADD A,40H	istruzioni per
CEF	6F		LD L,A	incrementare
CF0	7C		LD A,H	di 40H
CF1	CE 00		ADC A,0	la coppia
CF3	67		LD H,A	HL (riga sotto)
CF4	72	A3	LD (HL),D	
CF5	3E 20		LD A,20H	blank in Acc.
CF7	BA		CP D	cfr. con D
CF8	C4 04 0D		CALL NZ, RIT	esegui RIT solo se "*"
CFB	23		INC HL	nuova posizione
CFC	10 F6		DJNZ A3	B −1 e, se B = 0, torna ad A3
CFE	7D		LD A,L	se B = 0, invece
CFF	FE 7A		CP 7AH	cfr. L con 7AH (fine riga)
D01	20 D3		JR NZ,X	se non fine riga, torna a X
D03	C9		RET	fine routine PRINC
D04	C5	RIT	PUSH BC	inizio routine ritardo
D05	E5		PUSH HL	
D06	21 C0 0C		LD HL,R	
D09	46		LD B,(HL)	
D0A	CD 35 00		CALL KDEL	
D0D	10 FB		DJNZ −3	
D0F	E1		POP HL	
D10	C1		POP BC	
D11	C9		RET	fine routine PRINC

Una osservazione va anche fatta a proposito della comodità della istruzione CALL NZ, RIT all'indirizzo CF8: è sufficiente questo *richiamo condizionato* perchè, quando non si ha il blank, ossia in D c'è l'asterisco, il ritardo non abbia luogo. Istruttivo, vero?

Per concludere, richiamiamo per l'ennesima volta le modalità operative, dopo il caricamento in memoria, dei due ultimi programmi. Occorre il comando E, seguito dagli indirizzi C7C e, rispettivamente, CC1, ovvero le label KEYB1 e ONDAQUAD, rispettivamente.

Programma P. 4-4.

SCRITTA VIAGGIANTE

Scopo di questo, speriamo, divertente programmino è quello di far comparire sullo schermo una scritta mobile, che viaggia da destra verso sinistra. Un carattere alla volta fa capolino, poi, una volta completa, essa naviga tronfia su una riga poi comincia a scomparire a sinistra. Rispetto ad analoghe diciture mobili che si vedono alla televisione, questa ha l'inconveniente di una marcia a scatti, che la rende un poco buffa.

L'idea si basa sul fatto, già detto, che di ogni riga sulla video RAM solamente 48 posizioni sono visibili. Altre 16 *non visibili* ve ne sono a destra (cioè di indirizzo maggiore) ed altrettante sulla sinistra (di indirizzo cioè minore) e, a loro volta, poste a destra della riga superiore.

Questo almeno è quanto avviene per le righe diverse dalla "top".

Con riferimento alla riga il cui primo byte visibile ha indirizzo 088A, la situazione è illustrata qui sotto (con la lunghezza, in numero di byte, espressa in esadecimale).

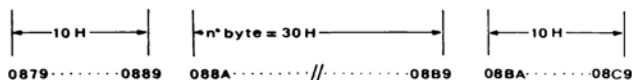



Figura 4-4

Prima soluzione (P. 4-4 a)

Il flow-chart d'assieme è riportato nella figura 4-5.

Per far partire la scritta si usa la solita routine del monitor KBD, seguita dal salto relativo −3 sulla condizione NC., per attendere che un tasto qualsiasi venga schiacciato (°).

(°) - Per la cronaca, nel NASCOM esiste anche una routine, detta CHIN, che attende in permanenza (cioè non si può uscire da essa senza che un tasto sia premuto). CHIN serve anche l'UART per fare i caricamenti da cassetta. Con CHIN non occorre il JR NC, −3.

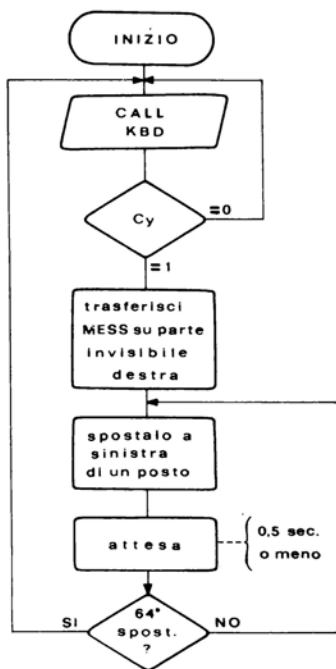



Figura 4-5 Scritta mobile (1ª versione).

Quando questo avviene, il programma provvede prima a trasferire MESS, che è una stringa di *sedici* caratteri contenuta in altra area di memoria (ovviamente non visibile) sulla parte invisibile destra (posizioni 08BA fino a 08C9). Quindi, da questà area, si fa lo spostamento a sinistra di un posto, poi si attende un tempo, ad es., di 0,4 ÷ 0,5 secondi (o anche meno o più, a seconda della velocità desiderata). Ripetendo questo procedimento per 64 volte (esadecimale 40H), la scritta viaggiante va infine a collocarsi esattamente nella zona non visibile sinistra (posizioni 0879÷0889)

A questo punto si ritorna a CALL KBD, affinchè, premendo ancora un tasto, si possa volendo far ripartire la scritta. Il giochetto dopo un pò naturalmente viene a noia, ma all'inizio può stupire gli amici.

Programma P. 4-4 a

Indirizzi	Contenuto	Label	Codice Assembly		Commenti
			ORG	C50H	
		KBD	EQU	69H	
		KDEL	EQU	35H	
C50 57	20 47 49	MESS	DEFM		'W GIACCAGLINI!ƀ
C54 41	43 43 41				
C58 47	4C 49 4E				
C5C 49	21 21 20				
C60	CD 69 00	INIZ	CALL	KBD	
C63	30 FB		JR NC,	INIZ	
C65	21 50 0C		LD HL,	MESS	trasferisci MESS su
C68	11 BA 08		LD DE,	08BAH	parte invisibile
C6B	01 10 00		LD BC,	10H	destra
C6E	ED B0		LDIR		
C70	06 40		LD B,	40H	totale caratt. (vis.+inv.)
C72	21 BA 08		LD HL,	08BAH	prepara l'
C75	11 B9 08		LD DE,	08B9H	uscita del 1° carattere
C78	C5	SPOST	PUSH	BC	salva Byte Counter BC
C79	E5		PUSH	HL	ed i puntatori HL
C7A	D5		PUSH	DE	e DE
C7B	01 10 00		LD BC,	10H	
C7E	ED B0		LDIR		fai scorrere MESS 1 posto
C80	06 50	ATT	LD	B,50H	attendi
C82	CD 35 00		CALL	KDEL	5 × 80 = 400 msec.
C85	10 FB		DJNZ	ATT	
C87	D1		POP	DE	ripristina puntatori DE
C88	E1		POP	HL	HL e
C89	C1		POP	BC	Byte Counter
C8A	1B		DEC	DE	
C8B	2B		DEC HL		
C8C	05		DEC	B	(uno spost. è stato fatto)
C8D	20 E9		JR	NZ,SPOST	se B=0, rifai SPOST
C8F	C3 60 0C		JP	INIZIO	se B=0, va a INIZIO

Vediamo di commentare la codifica.

Il programma ha inizio all'indirizzo C50 (poichè la versione che vedremo è un pò più compatta, questa soluzione non la mettiamo in coda al precedente programma P. 4-3). Qui si colloca una stringa di caratteri qualsiasi scritti nel solito codice, praticamente coincidente al 90% con l'ASCII, della ROM generatrice di caratteri che si è già detto. Per l'assemblatore, si usa il codice DEFM (DEFine Message), con il messaggio entro apici. La scritta è denominata con la label MESS (associata all'indirizzo C50). A titolo di esempio, si è previsto l'autoincensatorio: 'W GIACCAGLINI!! ƀ' dove la b minuscola con il gambetto tagliato simboleggia il blank. È infatti opportuno che il sedicesimo carattere sia tale, in quanto, come si comprende subito dopo un momento di riflessione, per il meccanismo di scorrimento che ab-

biamo illustrato la scritta viaggiante nel suo movimento lascia dietro di sé una scia di caratteri tutti uguali all'ultimo che, alla fine, riempiono interamente lo schermo.

Come si è già detto, inizialmente si mette: in HL, 0C50; in DE, 08BA e, nel Byte Counter BC, 10H (16 decimale). La LDIR provvede allora a spostare MESS sulla parte invisibile destra della riga.

Seguono le istruzioni: LD B, 40H; LD HL, 08BA e LD DE, 08B9, che, come è immediato capire, preparano il primo scorrimento per la fuoriuscita del primo carattere. Il valore messo in B in questo momento non è però ancora quello che serve a tale primo scorrimento. Infatti 40H, ossia 64 in decimale, serve a contare (con la solita tecnica ... del gambero) gli altrettanti scorrimenti necessari per esaurire il viaggio.

Si fa a questo punto il PUSH di BC, HL e DE. Questo è molto importante: i valori di questi puntatori e di B sono così *salvati* (°) e, dopo le istruzioni:

```
LD      BC,10H  (usa BC come Byte Counter vero e proprio)
LDIR
```

più quelle relative alla pausa di circa 400 msec. (istruzioni da C80 a C85), si torna a mettere in BC, HL e DE i valori prima salvati sullo stack, cioè, dopo il primo giro, 40H; 08BAH E 08B9H. In tal modo, decrementandoli tutti e tre (passando cioè, dopo il primo giro, rispettivamente a 3FH, 08B9H e 08B8H) si può riprendere un altro passetto a sinistra, con B che, col suo decrementare, "prende nota" dell'avvenuto spostamento precedente.

Questo lavoro di salvataggio e successivo recupero è reso indispensabile dal fatto che la LDIR ha l'effetto di alterare tutte e tre le coppie di coppie di registri. Quando B da 40H, a furia di scalare, arriva a zero il giochetto ha termine.

Si noti come funziona la LDIR (non, come di primo acchito poteva sembrare, LDDR!): con, ad esempio HL = 08BAH e DE = 08B9H (cioè in HL c'è un valore *inferiore di una unità* rispetto a quello contenuto in DE) il byte puntato da HL passa in quello *alla sua immediata sinistra,* puntato da DE, quindi HL e DE aumentano entrambe di una unità e si trasferisce il byte 08BB in 08BA ecc. Il tutto è schematizzato nella Fig. 4-6 a.
Visto che siamo in argomento di istruzioni tipo LDIR, LDDR, può essere interessante parlare di un altro uso della LDIR con il puntatore HL posto ad un valore inferiore di uno rispetto a DE. In questo caso, come dovrebbe risultare di immediata evidenza nella stessa figura, in b, si ottiene il riempi-

---

(°) - Ma al tempo stesso *mantenuti* (parliamo di HL e DE, B qui non interessa) nei corrispondenti registri. Per questo, si noti, qui *non si poteva usare a tale scopo la istruzione EXX*, perché lo scambio altera i vecchi valori.

mento del blocco (di lunghezza pàri al valore iniziale di BC) *con tutti caratteri uguali a quello contenuto nella posizione di memoria inzialmente puntata da HL.* Può usarsi ad esempio per azzerare tutti gli elementi di un insieme.

Il linguaggio Assembly il sorgente simbolico potrebbe presentarsi così:

```
LD   BC,LUNG
LD   HL,IND
LD   DE,IND + 1
LD   (HL),CAR
LDIR
```

Dove LUNG corrisponde alla lunghezza del blocco, mentre IND è l'indirizzo del primo byte, CAR è infine associata al carattere che si vuol replicare BC volte, tramite una EQU.
Si possono trattare blocchi enormi. A dire il vero però con blocchi non lunghissimi (fino a 256 byte!) è più corta la routine seguente:

```
        LD     B,LUNG
CICLO   LD     HL,IND
        LD     (HL),CAR
        DJNZ   CICLO
```

Il lettore potrà facilmente prendere da solo in considerazione gli equivalenti usi di LDDR, per esercizio.

Seconda soluzione (P. 4-4 b)

In modo più semplice che nella precedente, il trasferimento del blocco MESS vien fatto in altrettante posizioni di memoria video a partire dalla 08B9, cioè la *prima visibile.* Ciò fa si che il primo carattere faccia capolino subito, quindi si itera 64 volte il procedimento, ogni volta decrementando il puntatore della posizione iniziale di arrivo DE, mentre HL è sempre rimesso in corrispondenza all'inizio di MESS Si veda la Fig. 4-6, in c.

La codifica a questo punto non dovrebbe presentare difficoltà e ne lasciamo l'analisi al lettore.

Il programma è messo in coda al precedente, onde parte con l'indirizzo D12.

La scritta scelta è ora: "W IL MICRO Z-80ƀ" (altre possibili: "SALI E TABACCHIƀ" "ASINO CHI LEGGEƀ" ecc.)

Volendo far scorrere sullo schermo un numero maggiore di caratteri le cose a prima vista si complicano. Per esempio, volendolo fare con 32 con una tecnica simile a quelle viste si dovrebbe fare prima il trasferimento dei primi sedici caratteri facendoli viaggiare finchè non escono tutti, poi inizia-

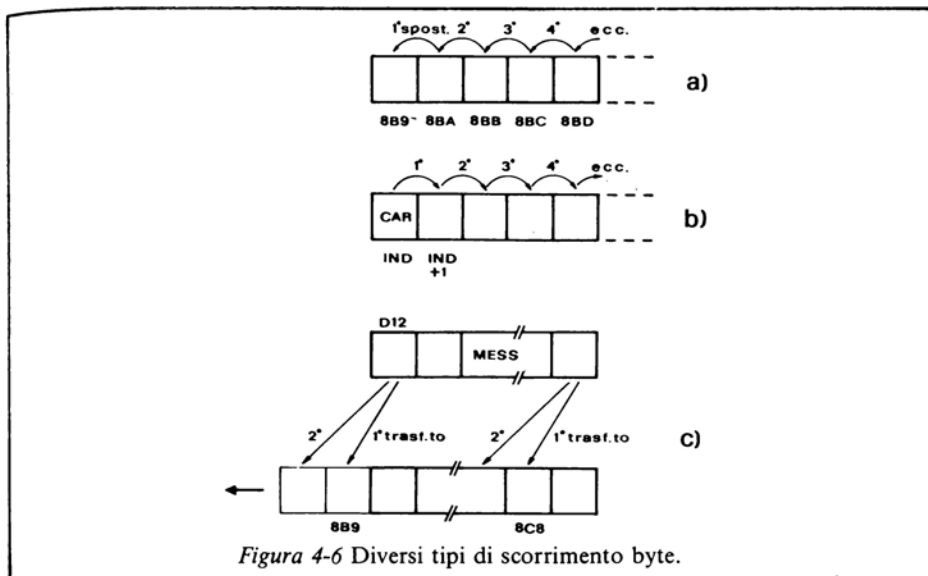



Figura 4-6 Diversi tipi di scorrimento byte.

Programma P. 4-4 b

Indirizzi		Contenuto			Label	Codice Assembl.	Commenti
D12	57	20	49	4C	MESS	DEFM	'W IL MICRO Z-80'
D16	20	4D	49	43			
D1A	52	4F	20	5A			
D1E	2D	38	30	20			
D22		CD	69	00	INIZ	CALL KBD	attendi un carattere
D25		30	FB			JR NC,INIZ	
D27		11	B9	08		LD DE,08B9H	
D2A		06	40			LD B,64	B = numero iterazioni
D2C		21	12	0D	RIPET	LD HL,MESS	
D2F		D5				PUSH DE	salva DE e
D30		C5				PUSH BC	BC (°)
D31		01	10	00		LD BC,16	BC = lunghezza blocco
D34		ED	B0			LDIR	trasf.to blocco in 08B9, poi in 08B8, 08B7 ecc.
D36		06	50		ATTESA	LD B,50H	ritardo di circa
D38		CD	35	00		CALL KDEL	80 × 5 = 400 msec.
D3B		10	FB			DJNZ ATTESA	
D3D		C1				POP BC	
D3E		D1				POP DE	
D3F		1B				DEC DE	1ª posiz. arrivo a sinistra
D40		10	EA			DJNZ RIPET	se B ≠ 0 rifai spost.to
D42		C3	60	0C		JP INIZ	con B = 0 torna a INIZ

(°) - Il salvataggio di HL ora non occorre: il valore di HL riprende sempre dall'inizio di MESS, con il salto a RIPET.

re a trasferire gli altri 16 facendo viaggiare l'intero blocco di 32 ecc. Per allenamento, il lettore potrebbe anche provare ad eseguire questo algoritmo.

Suggeriamo solo uno spunto formale di Assembly. Per definire le due parti del messaggio, si useranno le label MESS che punta all'inizio e MESS1 che si riferisce al primo byte del secondo gruppo di 16. La prima label serve allora per puntare inzialmente, con HL, al primo byte, sia nel trasferimento del blocco intero che in quello dei primi 16 byte (basta, nei due casi, assegnare un diverso valore a BC). MESS1 serve invece a trasferire il solo secondo gruppo.

Algoritmo generalizzato

Ma l'algoritmo migliore, che si presta senza difficoltà al viaggio di gruppi di caratteri di qualsivoglia lunghezza, è un altro.

Esso consta dei passi seguenti:

a) si carica il primo byte di MESS nell'ultima posizione della riga della video RAM; identifichiamo, per motivi di generalizzazione, con ULT tale posizione (per la riga finora considerata, si tratta di 08B9, v. sempre Fig. 4-4);

b) si fa shiftare *tutta la riga*, dopo il solito ritardo, chiamiamolo RITARD, verso sinistra; si noti che in tal modo non occorrono affatto aree invisibili e che, se LUNGRIG è la lunghezza della riga, i trasferimenti da compiere sono in numero di LUNGRIG - 1; l'unica cosa richiesta è che, inizialmente, la riga sia a blank;

c) si passa al secondo (poi al terzo ecc.) carattere di MESS e quindi si ripetono i passi a, poi b, per tutta la lunghezza del blocco: se questo supera la capienza della riga, ad un certo punto i primi caratteri scompaiono gradualmente sulla sinistra.
La codifica in Assembly potrebbe essere come alla pagina seguente.

Si osservi come il byte ULT costituisca un "punto di entrata" dei caratteri che man mano provengono da MESS. Le istruzioni da LD HL, PRIM + 1 a LDIR sono le analoghe di quelle già viste in precedenza ed attuano la tecnica illustrata in Fig. 4-6 a (al posto di 08B9 e 08BA ecc. qui vanno naturalmente messi 0889, 088A ecc.).

Quando, con B = 0, si supera l'istruzione DJNZ RIFAI, resta ancora da espletare, a piacere, il viaggio degli ultimi caratteri che ora riempiono, totalmente o parzialmente, lo schermo.

Una soluzione "brute force", come direbbero gli anglosassoni, consiste nell'allungare con 48 blank, tanta è la lunghezza dello schermo, il messaggio. Non è poi troppo, se si tiene conto che la coda di programma che temevamo così non è più necessaria, dato che alla fine questi 48 spazi vuoti riem-

Programma P. 4-4 c.

Label	Codice Assembly	Commenti
	LD B,LUNBL	carica in B lungh. blocco MESS
	LD HL,MESS	
RIFAI	LD DE,ULT	(es. ULT = 08B9H)
	LD A,(HL)	primo caratt. di MESS in
	LD (DE),A	ULT (imo) di riga TV
	CALL RITARD	opportuna routine di pausa
	PUSH BC	
	PUSH DE	
	PUSH HL	
	LD HL,PRIM + 1	PRIM = ind. 1° byte di riga
	LD DE,PRIM	(es. PRIM = 088A)
	LD BC,LUNRIG −1	es. LUNGRING − 1 = 2FH (30 − 1)
	LDIR	
	POP HL	
	POP DE	
	POP BC	
	INC HL	passa ad altro car. di MESS
	DJNZ RIFAI	Dec. B e, se = 0 ricomincia

piono lo schermo. Si può anche, con questo metodo che non necessita di aree invisibili, far apparire il messaggio vagante su una parte più limitata, diciamo 40, anzichè 48 posizioni.

Si diverta chi ci legge con sì ammirevole pazienza a codificare il tutto e non c'è ne voglia se gli abbiamo dato la soluzione più bella solo in ultimo: *per aspera ad astra ....*

A conclusione di questo capitolo dall'aspetto un po' frivolo possiamo dire che solo in apparenza abbiamo "giocato" con la nostra trappoletta.

In realtà, oltre ad aver appreso l'uso di tante istruzioni e routine che possono essere utilizzate in problemi serissimi, ad una analisi più accorta anche molti degli esempi visti sono imparentati, e a volte neanche tanto alla lontana, con problemi reali.

Ad esempio, il programmino dell'onda quadra presenta problemi molto simili a quelli che si hanno nei *temporizzatori* (dove il "duty cycle", ossia il rapporto, in un ciclo di lavoro, tra il tempo in cui un certo dispositivo è in azione e quello in cui esso è inattivo, è normalmente all'ordine del giorno).

L'ultimo lavoro fatto poi potrebbe addirittura trovare concreta applicazione presso i ... televisionàri.

Un ultimo esempio-gioco orientato abbastanza chiaramente ad una applicazione specifica è quello del LUCCHETTO ELETTRONICO. (Programma P. 4-5).

In un'area di memoria è contenuta la *combinazione* (mettiamo 9 byte), ossia un insieme di cifre e magari, per rendere più difficile la vita ai ladri, anche lettere. Ogni volta che si batte un tasto, il programma lo confronta successivamente con la 1ª, la 2ª, ecc. posizioni di una tabella COMBIN. Se la

battuta è conforme a tale contenuto, si passa avanti, in caso contrario, viene lanciato un allarme.

Il lettore non dovrebbe aver difficoltà a svolgere per proprio conto il problemino. Confronti comunque poi con la soluzione, in solo Assembly, che qui di seguito proponiamo.

In questa, si sono utilizzate all'inizio le label: NOVE per il numero 9 che è la lunghezza comune a COMBIN come pure a MESAL e MESOK, che sono i messaggi di allarme e di consenso. Inoltre INIRIG si riferisce ad uno dei primi byte della "top row" su cui una delle due scritte dovrà comparire.

Osservazioni:

ci limitiamo a parlare del giro di registri puntatori. Inizialmente in DE è posto MESALL. Così facendo, quando la comparazione tra cifra battuta e quella in COMBIN puntata da HL dà errore la JR NZ,MESS salta in fondo, ove si hanno istruzioni che scrivono l'allarme. Se invece Z = 1, l'allarme non è lanciato e si prosegue il confronto fino a che B, da nove come era all'inizio, arriva a zero. In tal caso in DE è posto MESOK.

Programma P. 4-5

Label	Codice Assembly	Commenti
NOVE	EQU 9	
INIRING	EQU 0BCDH	sulla "top row"
COMBIN	DEFM '........'	combinazione (a piacere)
MESALL	DEFM 'AL LADRO!'	
MESOK	DEFM 'TUTTO OK!'	
LUKETT	LD HL,COMBIN	inizio programmino
	LD B,NOVE	
	LD DE,MESALL	preparati allarme!
KEYB	CALL KBD	
	JR NC,KEYB	
	CP (HL)	cfr. con caratt. in memoria
	JR NZ,MESS	diverso? lancia allarme!
	INC HL	passa ad altra cifra combinazione
	DJNZ KEYB	attendi nuova battuta ecc.
	LD DE,MESOK	se tutte 9 esatte, MESOK
MESS	EX DE,HL	MESALL o MESOK in HL
	LD BC,NOVE	
	LD DE,INIRIG	scrivili a partire da
	LDIR	INIRIG
	HALT	oppure istruzione di ritorno al monitor.

In tutti i casi si arriva a MESS con, in DE, l'indirizzo giusto per il messaggio, quale che sia, da lanciare. A questo punto, la EX DE, HL serve a porre DE in HL, in quanto la solita trafila che termina con LDIR richiede in HL l'indirizzo iniziale da cui copiare.

CAPITOLO 5

SOTTOPROGRAMMI VARI DI CONVERSIONE TRA CODICI

I testi in commercio che riguardano microprocessori abbondano di esempi di tipo aritmetico, anche alquanto complicati e sacrificano troppo, a nostro avviso, quelli applicativi. Non si può comunque negare l'importanza di tali cose e pertanto dedichiamo un capitolo alle procedure tra le più noiose, ma purtroppo necessarie nella manipolazione dei dati, ossia quelle di conversione da un codice all'altro.

Gli esempi che stiamo per fare sono stati elaborati in modo autonomo.

Pertanto, nella letteratura specializzata, può anche capitare che ce ne siano di migliori, per esempio come compattezza. La cosa non dovrebbe scandalizzare, dato che in fondo tale letteratura è appunto disponibile per essere ... copiata, ma noi volevamo esercitarci per nostro conto (in un paio di casi, c'è persino capitato di verificare che la nostra soluzione era pressochè identica nell'impostazione o addirittura migliore di quelle proposte dalle sacre fonti).

Iniziamo con la più comune, la conversione da binario a BCD.

Programma P. 5-1

CONVERSIONE BINARIO-BCD: 1° METODO

L'algoritmo è illustrato nel diagramma a blocchi della Fig. 5-1.

Il numero binario, *a 16 bit,* da convertire, chiamato BIN, si trova nelle celle di memoria C50 ÷ C51. Il numero equivalente decimale, in codice BCD, dovrà essere sistemato nelle posizioni C52, C53 e C54. In entrambi i casi, il digit più significativo è quello contenuto nel byte di indirizzo minore (cioè C50 e C52, rispettivamente).

Il metodo consiste nel sottrarre ripetutamente, al numero binario prima 10.000, poi 100, ove 10.000 e 100 sono naturalmente codificati in binario (e, quindi, in notazione esadecimale sono espressi rispettivamente da 2710 e 64). Si trova così inizialmente quante volte 10.000 è contenuto nel numero da convertire. Questo risultato, contenuto nell'Accumulatore, che, si ricorda, ha *otto* bit, per forza di cose ha i primi 4 bit nulli, con gli altri quattro che costituiscono una cifra $\leqslant 6$. Infatti il massimo numero binario a 16 bit vale 65.535, in decimale.

La prima operazione ci dà pertanto il byte BCD di ordine maggiore, già corretto. Nella seconda fase però, con la sottrazione ripetuta di 100, non si

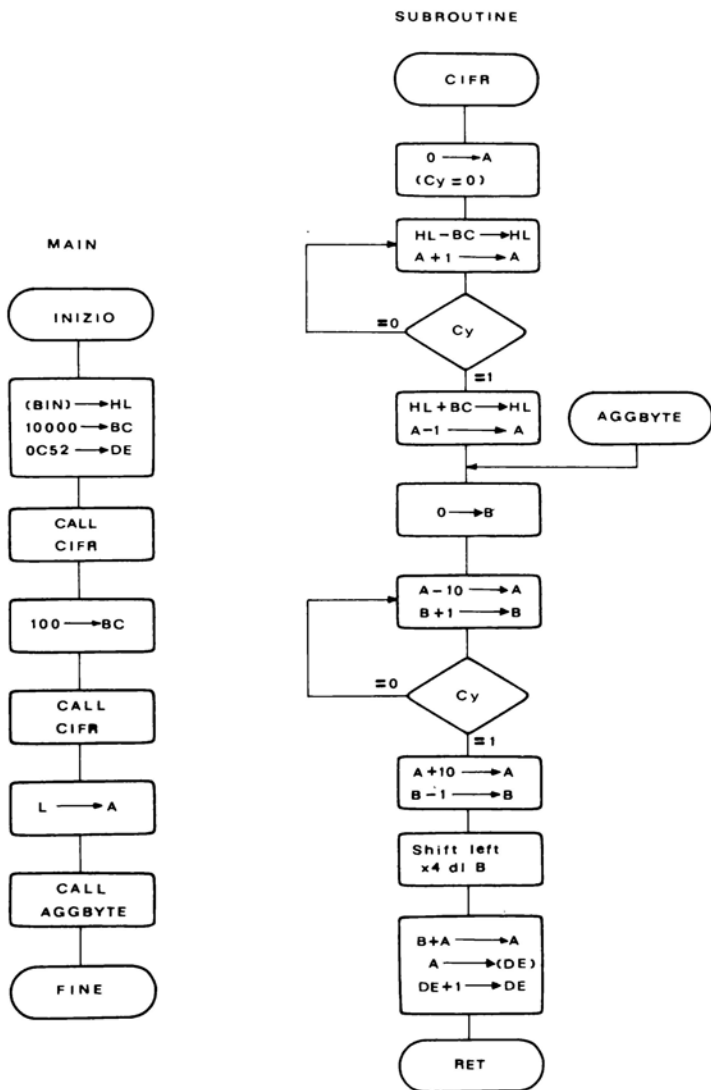



Figura 5-1 Conversione in BCD con metodo aritmetico.

ottiene con la stessa tecnica, tranne eccezioni, il byte BCD intermedio (°) da porre all'indirizzo C53, quindi occorre trasformarlo ulteriormente, prima di trasferirlo in memoria.

Spieghiamoci con un esempio numerico: sia da convertire A2F1H (in decimale vale 41713); sottraendogli ripetutamente 2710H (cioè 10.000) si trova 04H che ovviamente coincide con il valore BCD. Resta il numero 6B1 (in decimale, ovviamente, vale 1713) e sottraendo ripetutamente a quest'ultimo 64H (cioè 100D) si ottiene nell'accumulatore l'esadecimale 11H, che in decimale vale 17, ma non è ancora la corretta rappresentazione in BCD.

Per ottenerla, occorre sottrarre ripetutamente 10 (esadecimale 0AH) lavorando anche opportunamente sul semibyte.

A questo punto vediamo più da vicino il flow chart.

Si ha un programma principale, MAIN (inizia dall'indirizzo C73, v. codifica più avanti) e una sobroutine a due entrate, l'una CIFR chiamata per due volte, mentre l'altra entrata è alla label AGGBYTE.

MAIN: dopo aver caricato il valore di BIN in HL (si osserverà nella codifica che, avendo preferito porre il byte più "pesante" in C50, mentre la istruzione LD HL, (BIN) carica (0C50H) in L e (0C51) in H, si deve poi scambiare H con L) si pone in DE l'indirizzo BCD, cioè C52H e in BC 2710H, cioè, ripetiamo, 10.000. Viene chiamata la routine CIFR, si pone poi in BC il valore 64H, cioè 100D, evocando ancora CIFR. Infine si chiama AGGBYTE, che è la parte terminale di CIFR, dopo aver posto L nell'Accumulatore.

Routine CIFR. Con XOR A si azzera l'Accumulatore, resettando il Carry flag: questo è un provvedimento prudenziale anche generale. Per esempio, possiamo ragionevolmente supporre che il nostro programmino sia a sua volta una subroutine e il giorno in cui lo si usasse come tale potrebbe capitare di entrare in esso con Cy = 1.

Ora noi vogliamo sfruttare l'istruzione di aritmetica a 16 bit SBC HL, BC (non possiamo usarne una del tipo SUB HL,BC perchè ... non esiste!) e così, con Cy = 1, rischiamo di sottrarre 10.001 o 101, anzichè 10.000 e 100 (°°).

Ma senza pensare a cose così remote, è nel corso dello stesso CIFR che, come si vedrà, Cy viene settato.

(°) - Per chi non l'avesse ben chiaro, stiamo implicitamente riferendoci alla forma "impaccata", con la quale si hanno due cifre BCD per byte.

(°°) - Programmare, dunque, significa essere previdenti e prevederne sempre una più del diavolo.

Dunque, si toglie BC da HL e si incrementa A ripetutamente finchè Cy = 1. A questo punto A contiene il numero (*binario!*) di volte che (BIN) contiene 10.000 (oppure, più avanti, 100), *più una unità,* mentre la coppia HL ha un numero in complemento a 2 (a 16 bit), in quanto gli è stato sottratto il contenuto di BC una volta di troppo. Poco male, non c'è che da togliere 1 da A e aggiungere BC ad HL, per rimettere le cose a posto. Il lettore accorto avrà subito riconosciuto una notevole somiglianza di questo con il procedimento della divisione per differenze ripetute visto fin dal Cap. 1.

A questo punto incomincia AGGBYTE, con l'azzeramento di B e la ripetuta sottrazione di 10 (cioé l'esadecimale A) dall'Accumulatore, con il registro B che funge da contatore. Quando Cy si accende, in modo perfettamente analogo a quanto già visto sopra, si rimettono a posto i contenuti di A aggiungendogli 10 (cioé, sempre, AH) e di B incrementandolo di uno.

Riferendoci all'esempio lasciato in sospeso nella parte introduttiva, quando A, dopo ripetute sottrazioni di 100 da HL contiene 11H, ossia 17D, per ottenere la *corretta* codifica BCD, con l'esecuzine di AGGBYTE si ricava, con reiterate sottrazioni di AH, in Accumulatore il valore 7 ed in B il valore di 1 (°). A questo punto con quattro istruzioni tipo SLA (Shift Left Arithmetic) applicate a B se ne trasforma il contenuto in 10H, che, aggiunto ad A, dà finalmente 17H. Questo dato è caricato in (DE), in quanto DE è (lo avevamo lasciato in sospeso) il puntatore dei byte del risultato, quindi con l'INC DE si punta alla nuova locazione.

Tornando un momento al MAIN, si osservi che la routine CIFR è chiamata in gioco interamente anche quando BC = 10.000. In questo caso, come si è visto, la parte "caudale" AGGBYTE sarebbe superflua, dato che l'Accumulatore contiene già un risultato corretto (04H nel nostro esempio numerico). Tuttavia l'esecuzione di AGGBYTE non dà (e come potrebbe?) errori (ad es.: si ottiene A = 04 e B = 0 e quindi ancora B = 0 dopo i quattro shift e ancora A + B = 04); c'è solamente una lieve perdita di tempo, compensata però dalla compattezza del programma.

Quando infine, dopo aver sottratto ripetutamente prima 10.000, poi 100, si debbono determinare le ultime due cifre, l'equivalente esadecimale di esse *si trova per forza in L* (e H = 0).

Infatti il massimo valore decimale residuo possibile è, a questo punto, 99 che, direi, *a fortiori* "ci sta" in esadecimale in un byte (il cui valore binario massimo vale FFH = 256D).

(°) - Speriamo che il lettore benevolo non ce ne vorrà se l'esempio non è felicissimo, in quanto si fa solo *una* iterazione ...

Anziché allora procedere, per meccanica analogia, con il trasferimento di 1 in BC e la CALL di CIFR, si mette direttamente L in A e si chiama la routine dalla label AGGBYTE.

Programma P. 5-1

Indir.	Contenuto	Label	Cod. Assembly		Commenti
			ORG	C50H	
C50	00 00	BIN	DEFW	OH	v. nota 1
C52	00 00 00	BCD	DEFS	3	riserva 3 byte (v. nota 1)
C55	AF	CIFR	XOR	A	A = 0; Cy = 0
C56	ED 42		SBC	HL, BC	
C58	3C		INC	A	
C59	30F B		JR	NC, −3	ritorna a SBC HL, BC se Cy=0
C5B	09		ADD	HL, BC	con Cy=1 rimetti a posto HL
C5C	3D		DEC	A	e Acc.
C5D	06 00	AGGBYTE	LD	B,0	
C5F	D6 0A		SUB	10	togli 10 ad Acc.
C61	04		INC	B	
C62	30 FB		JR	NC, −3	finché Cy=0 ripeti A − 10 ecc.
C64	C6 0A		ADD	A,10	con Cy=1 aggiusta A
C66	05		DEC	B	e B
C67	CB 20		SLA	B	esegui
C69	CB 20		SLA	B	quattro
C6B	CB 20		SLA	B	shift aritmetici
C6D	CB 20		SLA	B	di B
C6F	80		ADD	A, B	
C70	12		LD	(DE), A	carica un byte di BCD
C71	13		INC	DE	poi punta al seguente
C72	C9		RET		fine CIFR (e AGGBYTE)
C73	2A 50 0C	INIMAIN	LD HL,	(BIN)	voce BIN in HL: inizio MAIN
C76	7C		LD A,	H	inversione di
C77	65		LD H,	L	H con
C78	6F		LD L,	A	L
C79	11 52 0C		LD DE,	BCD	punta DE a 1° byte BCD
C7C	01 10 27		LD BC,	10000	(esadec. 2710H)
C7F	CD 55 0C		CALL	CIFR	
C82	01 64 00		LD BC,	100	(esad. 64 H)
C85	CD 55 0C		CALL	CIFR	
C88	7D		LD A,	L	
C89	CD 5D 0C		CALL	AGGBYTE	
C8C	31 33 0C		LD SP,	C33H	ritorno al
C8F	C3 59 03		JP	359H	monitor
			END		

Note: 1) gli zeri messi in 0C50 fino a 0C54 sono riempitivi (qualsiasi) per la sola fase di compilazione; nell'esecuzione, si metterà in C50 e C51 il numero da convertire (col comando M).
2) per l'esecuzione, dopo il caricamento ed il "." NL fare "E", seguito dall'indirizzo C73 di INIMAIN, poi verificare, con il comando M, la correttezza del dato convertito in BCD: es: mettere in C50 - C51 F1A2H = 61858D; in C52-C53-C54 vi deve essere, al termine, 06H, 18H e 58H.

La codifica riportata alla pagina precedente dopo la amplissima e dettagliata spiegazione che abbiamo fornito non richiede alcun commento (°).

Programma P. 5-2 a (v. pag. 97-98)

CONVERSIONE IN BCD CON "DOUBLE AND DUBBLE"

Come qualcuno forse saprà, il metodo consiste nel prendere ad uno ad uno i bit del numero binario da convertire, *dal più alto al più basso* e, ognivolta che i passa al successivo, raddoppiare ed aggiungere il nuovo bit.

Esempio: sia il numero binario 10110, i passi successivi richiesti dall'algoritmo sono illustrati qui di seguito:

Successivi bit	Succ. n° decimale
1	1
0	1 × 2 + 0 = 2
1	2 × 2 + 1 = 5
1	5 × 2 + 1 = 11
0	11 × 2 + 0 = 22

Qui ci conviene illustrare direttamente il programmino, senza il flow chart (dato il suo eccessivo tecnicismo, che mal si presta ad essere tradotto graficamente).

Il dato binario a 16 bit stavolta lo si suppone presente nella coppia registri DE, mentre HL è un puntatore verso tre byte di memoria C50, C51 e C52 destinati all'equivalente decimale.

Il programma comincia all'indirizzo C6F, mentre la routine interna, qui denominata SOTPRG, si trova all'indirizzo C53. Questa tecnica, già vista nel caso precedente, di mettere i sottoprogrammi all'inizio, è, ripetiamo, vantaggiosa quando si lavora direttamente in linguaggio macchina, in quanto, quando si arriva alle diverse CALL, i relativi indirizzi sono già noti. Inoltre, se avviene, come in questo caso, che la routine sia un po' il cuore del programma, è abbastanza più logico che se ne parli per prima. Facciamolo.

(°) - Merita invece una noticina un dettaglio del flow, ossia le scritte CIFR e AGGBYTE nei simboli di limite e nel punto in cui inizia la parte terminale di CIFR. Peraltro, si tratta di particolari di evidenza immediata, di cui va solo sottolineata l'efficacia.

Routine SOTPRG. Commentiamo riga per riga il "sorgente" Assembly.

- LD B, 8 poi LD HL, BCADR + 2: si inizia caricando 8 nel registro B e, in HL, l'indirizzo BCADR +2, ossia l'indirzzo C52 del byte meno significativo del risultato decimale; questo come si vedrà all'inizio del "main", è da principio *riempito con zeri.*

- RLC C più JR NC, GIU: si ruota a sinistra di un bit il registro C che contiene, nella prima chiamata di SOTPRG, il byte più elevato del numero da convertire, mentre nella seconda chiamata conterrà invece E. Con l'uso di C come registro di "comodo" (o, come si dice, anche di "deposito" o "transito") si è così evitata la necessità di scrivere due diverse routine, una per la conversione di D, l'altra per la stessa operazione applicata ad E. L'istruzione RLC mette nel carry flag Cy il primo, poi, durante lo svolgimento del loop di otto cicli, il secondo, il terzo ... l'ottavo bit di C. Si testa Cy e, se esso è zero (condizione NC), si fa un salto relativo alla label GIU.

- LD A, (HL); INC A; DAA; LD (HL), A. Con Cy = 1 si carica in modo indiretto tramite HL il contenuto di C52 in A, lo si aumenta di una unità e si opera l'aggiustamento binario/BCD mediante l'istruzione DAA (cioé Decimal Adjust Accumulator), poi il contenuto di A è messo di nuovo in (HL). La necessità del DAA può apparire superflua e "curiosa" al primo passaggio, cioé quando A passa da zero a 1, ma in seguito essa è indispensabile.

Supponiamo per esempio che, portando avanti il procedimento di conversione, ad un certo punto il byte C52 contenga 19 (ossia, bit per bit: 00011001).
Con l'incremento (in binario!) di 1 si passa a 1AH (binario 00011010). La DAA allora serve appunto a riportare l'accumulatore al valore corretto 20.

- DEC HL; LD A, 0; ADC A, (HL); DAA: questa serie di istruzioni, ripetuta per due volte, serve a passare al successivo byte (cioé prima C51, quindi C50) aumentandone il valore di 1, se in precedenza c'è stato riporto.

Questo può avvenire ogni volta che il byte C52 (o C51) conteneva 99: con l'aggiunta di 1 si passa (nell'Accumulatore) a 9A senza riporto, che viene però generato dalla DAA, che produce 00 con Cy = 1 (come è logico e giusto e secondo quanto il lettore diffidente potrà verificare consultando il manuale dello Z80).

Osserviamo: caricare zero in A per poi aggiungergli l'eventuale riporto più (HL) può sembrare un po' ozioso, ma è la cosa più pratica per tener

conto automaticamente del caso in cui c'è o non c'è il riporto.
Una soluzione forse un pelo più veloce, consistente nell'evitare di aggiungere zero ai successivi byte quando Cy = 0 potrebbe essere la seguente (riportiamo per comodità più codici Assembly in una stessa riga):

```
        JR   NC, GIU        DEC   HL
        LD   A, (HL)        INC   A      DAA LD (HL), A
        JR   NC, GIU -1     DEC   HL
        LD   A, (HL)        INC   A      DAA LD (HL), A
        INC  HL             INC   HL
GIU     DEC  B              eccetera (cfr. con la codifica completa).
```

Essa risulta un po' meno compatta di quella già adottata.
Riepilogando, abbiamo visto che, quando il bit estratto da C era uguale a 1, lo si è aggiunto, *in decimale*, al precedente valore contenuto in C50, C51 e C52. Se invece tale bit era zero, si saltava (evitando, qui almeno, l'aggiunta di zero) alla label GIU, cui comunque siamo ora arrivati.
Prima di tale label sono le istruzioni INC HL e ancora INC HL che servono, prima dell'eventuale uscita dalla routine che stiamo per vedere, a rimettere a posto in HL l'indirizzo C52 del byte più "leggero".

- GIU DEC B; RET Z: ossia, se togliendo uno a B si ottiene B = 0, siamo evidentemente alla fine della routine SOTPRG, in quanto abbiamo estratto poc'anzi l'ultimo bit (LSB) aggiungendolo, per l'ultima volta, al decimale convertito.
  Abbiamo in sostanza finora descritto la fase *"dubble"*. Vediamo ora la *"double"*.

- LD A, (HL); ADD A, A; DAA; LD (HL), A: il raddoppio del numero fino ad ora trovato ha, come istruzione chiave, ADD A, A che aggiunge A a se stesso. Al solito, DAA opera la correzione BCD.

Vediamo due esempi numerici di come operano queste istruzioni. Primo esempio: supponiamo che il dato, *decimale* dopo tutte le sistematiche DAA fatte in precedenza, sia 48; facendone il raddoppio si ottiene il valore 90H, col DAA ci si riporta al corretto decimale 96 (meglio, si dovrebbe dire: *valore BCD* = 96H = 10010110B). Secondo esempio: per raddoppiare 99, prima si ottiene: (2 × 99) H = 32 H con riporto Cy = 1 poi la DAA aggiusta l'Accumulatore in 98 e conserva il riporto Cy = 1.

- LD A, (HL); ADC A, A; DAA; LD (HL): precedute da DEC HL (per passare al successivo byte di ordine maggiore) queste istruzioni, ripetute due volte per i byte che restano, servono a completare il raddoppio. Si noti l'ADC A, A resa necessaria dal fatto che ora c'è da tener conto dell'eventuale Carry.

- JP SOTPROG + 2: a questo punto, non resta altro da fare che riprendere dall'indirizzo C55, cioé con il caricamento di C53 in HL e la rotazione di un altro bit di C ecc.

Programma chiamante. Compresa ora la funzione di SOTPRG, vediamo il main, che in questo caso a dir il vero non è ... un vero e proprio main. Infatti abbiamo concepito questo programmino CONVER come una routine di utilità da caricare eventualmente su EPROM per arricchire il firmware o simili tanto è vero che CONVER è fatto terminare con una RET.

Si comincia con il già preannunciato azzeramento dei tre byte C50, C51 e C52: dopo il solito XOR A, che azzera l'Accumulatore, questo contenuto è trasferito in (HL), con HL fatto inizialmente uguale a C50 (cioé BCDADR). Con due INC HL seguiti da LD (HL), A si completa l'opera.

A costo di annoiare mortalmente i lettori piu sagaci, ribadiamo che questo modo di procedere è più compatto: infatti una istruzione tipo LD (HL), 0 avrebbe richiesto 2 byte anziché l'unico di LD (HL), A.
Già che siamo in argomento, accenniamo al fatto che, dovendo fare l'azzeramento di più di tre byte conveniva un piccolo loop. Il lettore provi a farlo per suo conto, poi confronti con la seguente soluzione (prevista per 10 byte):

```
        XOR   A      LD HL, ADDR        LD B, AH
CICLO   LD    (HL),A  INC HL            DJNZ CICLO        (°)
```

La conclusione del chiamante non presenta difficoltà, essendo stata anche anticipata: si carica in C, registro di "comodo" prima D, ossia i bit più "pesanti" poi si fa appello a SOTPRG, quindi si ripete la medesima proceduta con i bit di ordine inferiore che si trovano in E (si ricorda che il metodo "double/dubble" procede partendo dai bit più alti come ordine).

Programma P. 5-2

Indir.	Contenuto	Label	Codice Assembly	Commenti
			ORG C50H	
C50		BCDADR	DEFS 3	dato BCD convertito
C53	06 08	SOTPRG	LD B, 8	inizio subroutine
C55	21 52 0C		LD HL, BCDADR +2	punta byte dext.
C58	CB 01		RLC C	estrai bit
C5A	30 12		JR NC, GIU	(salto relat. +20)
C5C	7E		LD A, (HL)	inizio "dubble":

(°) - Questa disposizione di più istruzioni su una riga non è solo un espediente tipografico, ma si ha anche con compilatori più potenti.

Indir.	Contenuto	Label	Cod. Assembly	Commenti
C5D	3C		INC A	aggiunta di 1 al decimale
C5E	27		DAA	
C5F	77		LD (HL), A	
C60	2B		DEC HL	byte a sinistra
C61	3E 00		LD A, 0	A = 0 con Cy *immutato*
C63	8E		ADC A, (HL)	
C64	27		DAA	
C65	77		LD (HL), A	
C66	2B		DEC HL	ultimo byte a sin.
C67	3E 00		LD A, 0	
C69	8E		ADC A, (HL)	
C6A	27		DAA	
C6B	77		LD (HL), A	"dubble" completo
C6C	23		INC HL	ripristina puntamento su
C6D	23		INC HL	byte di dext.
C6E	05	GIU	DEC B	altro bit?
C6F	C8		RET Z	se NO esci
C70	7E		LD A, (HL)	inizio "double"
C71	87		ADD A, A	
C72	27		DAA	
C73	77		LD (HL), A	
C74	2B		DEC HL	
C75	7E		LD A, (HL)	
C76	8F		ADC A, A	
C77	27		DAA	
C78	77		LD (HL), A	
C79	2B		DEC HL	
C7A	7E		LD A, (HL)	
C7B	8F		ADC A, A	
C7C	27		DAA	
C7D	77		LD (HL), A	
C7E	C3 55 0C		JR SOTPRG +2	da capo, saltando LD B,8
C81	AF	CONVER	XOR A	A = 0
C82	21 50 0C		LD HL, BCDADR	quindi
C85	77		LD (HL), A	azzera
C86	23		INC HL	BCDADR e
C87	77		LD (HL), A	seguenti
C88	23		INC HL	
C89	77		LD (HL), A	
C8A	4A		LD C, D	D in comodo
C8B	CD 53 00		CALL SOTPRG	converti byte più pesante
C8E	4B		LD C, E	E in comodo
C8F	CD 53 0C		CALL SOTPRG	converti byte più leggero
C92	C9		RET	fine routine CONVER

A questo punto, proponiamo al lettore un semplice *esercizio*: creare un programma chiamante la subroutine CONVER, testé scritta, che provvede alla conversione in BCD di un dato contenuto nelle posizioni di memoria C93 e C94 (per comodità di caricamento, abbiamo previsto il proseguimento a partire dall'ultima posizione di CONVER). Supponiamo che, anziché seguire la convenzione dello Z80 per cui, in una voce di due byte, quello di ordine inferiore precede l'altro, il dato binario a 16 bit qui sia scritto in modo opposto. *Soluzione:*

Indir.	Contenuto	Label	Cod. Assembly	Commenti
C93	00 00	DATOBIN	DEFW 0	
C95	ED 5B 93 0C		LD DE, DATOBIN	
C99	7A		LD A, D	scambio tra
C9A	53		LD D, E	D ed
C9B	5F		LD E, A	E
C9C	CD 81 0C		CALL CONVER	
C9E	31 33 0C		LD SP, C33H	solite istruzioni di
C9E	C3 59 03		JP 359H	ritorno al monitor
			END	

A dire il vero, era più compatto fare: LD HL, DATOBIN poi LD D, (HL) e, dopo INC HL, LD E, (HL), con sei byte occupati contro i sette della soluzione sopra. Con una coda del genere, si può fare, con modalità operative che ormai dovrebbero essere chiare, il collaudo del programma. Il lettore potrà anche eseguire lo "stepping" per controllare passo passo come la routine estrae i successivi bit di DE (posti in C ecc.) in Cy ecc. Qui è allora doveroso ricordare che Cy è, nel registro F, l'ultimo a destra. Gli altri sono: S = bit 7; Z = bit 6; H = bit 4; P/V = bit 2 e N = bit 1 (°). Questo esercizio richiede molta pazienza, in quanto sullo schermo la situazione dei vari registri è espressa in esadecimale, mentre qui occorrono anche i singoli bit (per Cy comunque la cosa è facile: basta controllare che il dato AF sia pari o dispari per dedurre Cy = 0 oppure = 1).
Esso è comunque molto istruttivo.

Programma P. 5-3

CONVERSIONE IN BCD CON DOUBLE/DUBBLE: VERSIONE COMPATTA

La soluzione precedente è stata elaborata, diciamo, artigianalmente preoccupandosi soprattutto di seguire correttamente l'algoritmo, senza badare troppo alla compattezza. Dopo vari tentativi, siamo riusciti a realizzare una versione più economica in termini di occupazione di memoria.

(°) - I flag H e N servono allo Z80 in certe sue operazioni. La loro importanza per il programmatore è talmente modesta che il linguaggio Assembly neanche li cita.

Il relativo flow chart è nella figura 5-2 e la codifica a pag. 102.

Esso prevede tre routine (la prima è P. 5-3 medesimo) "nidificate" (in nglese "nested") l'una nell'altra: BCDCONV che, per due volte, chiama la SOT, la quale, a sua volta, per tre volte di seguito chiama una subroutine

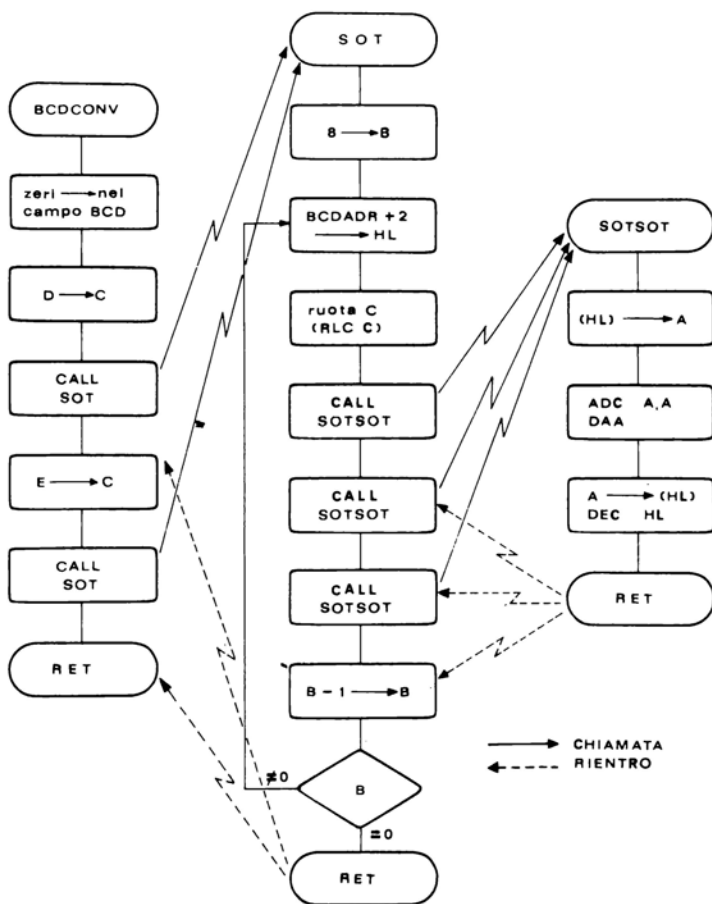



Figura 5-2 Conversione in BCD con "bouble & dubble".

più interna denominata SOTSOT. Le "saette" a linea intera e tratteggiata evidenziano questi passaggi. Quanto all'algoritmo, esso in sostanza sfrutta razionalmente l'istruzione ADC A, A ossia: aggiungi all'Accumulatore se stesso con l'aggiunta del riporto, sia esso zero o 1.

La sobroutine principale, BCDCONV, è identica a quella del caso precedente.

La SOT, come la SOTPRG del programma visto dianzi, inizia con B = 8 ed HL posto al valore C52, ossia quello del byte più basso di BCDADR, quindi si ruota circolarmente a sinistra il registro C. A questo punto, interviene SOTSOT. Vediamo come questa opera: si inizia con il trasferimento in A del valore contenuto nel byte puntato da HL, cioé inizialmente C52. Esso, come pure gli altri due a sinistra, la prima volta è zero (v. routine principale BCDCONV), quindi la istruzione successiva ADC A,A fa ... il doppio di zero cioè (indovinate un po'?) zero *ed aggiunge il Carry che è il bit appena estratto* dal registro C. In seguito, in C52 c'è il digit *decimale* (reso sempre tale grazie ai sistematici DAA) fino a quel momento convertito.

Anche in questo caso, il raddoppio più il riporto Cy servono naturalmente allo scopo, cioè seguire il criterio del "double and dubble". La DAA, che segue ogni volta, metodicamente mette le cose a posto dal punto di vista decimale. Seguono le istruzioni A trasferito in (HL) e DEC HL, con le quali prima si mette in C52, al primo giro, il digit convertito parzialmente, poi si passa, con la ri-chiamata di SOTSOT, a fare lo stesso lavoro sul byte C51, poi C50.

Al solito, il lettore scrupoloso potrà convincersi della bontà del metodo seguendo sulla carta, passetto per passetto, le varie istruzioni fingendo di essere lo Z80. Lasciamo anche alla sua proverbiale pazienza la verifica del programmino codificato (anzi, prima di andarlo a vedere, provi a codificarsi per suo conto il flow di Fig. 5-2): dato anche l'esempio precedente, la cosa non deve a questo punto presentare difficoltà. Facciamo infine notare che in teoria la triplice "subchiamata" di SOTSOT poteva anche essere evitata.

Al posto di tre CALL SOTSOT si sarebbe potuto fare, pedissequamente:

```
LD A, (HL)   ADC A, A   DAA   LD (HL),A   DEC HL
LD A, (HL)   ADC A, A   DAA   LD (HL),A   DEC HL
LD A, (HL)   ADC A, A   DAA   LD (HL),A
```

sarebbero occorsi, 14 byte (un ulteriore DEC HL è infatti superfluo). Con la soluzione proposta ci vogliono invece: $3 \times 3 + 6 = 15$ posizioni (3 volte CD 66 0C per la chiamata di SOTSOT più i 6 byte di essa, dato che si deve comprendere la RET). Quindi abbiamo fatto un (modestissimo) sacrificio all'eleganza strutturale, non solo, ma, così facendo, il programma è più agevolmente generalizzabile al caso di numeri grandi (ove addirittura converrà inserire la SOTSOT in un "loop" con decontatore posizionato inizialmente al valore corrispondente al numero di byte del numero decimale).

La maggior compatezza di questa versione è chiara: solo 46 byte contro i più di 60 degli altri due programmi precedenti.

Programma P. 5-3

Indir.	Contenuto	Label	Cod. Assembly	Commenti
			ORG C50	
C50	00 00 00	BCDADR	DEFS 3	
C53	06 08	SOT	LD B, 8	a
C55	21 52 0C	DINUOVO	LD HL,DCBADR +2	buon
C58	CB 01		RLC C	intendi-
C5A	CD 66 0C		CALL SOTSOT	tor
C5D	CD 66 0C		CALL SOTSOT	poche
C60	CD 66 0C		CALL SOTSOT	ciance
C63	10 F0		DJNZ DINUOVO	(v. testo
C65	C9		RET	e
C66	7E	SOTSOT	LD A, (HL)	anche
C67	8E		ADC A, A	programma
C68	27		DAA	P. 5-2)
C69	77		LD (HL), A	
C6A	2B		DEC HL	
C6B	C9		RET	
C6C	AF	BCDCONV	XOR A	
C6D	21 50 OC		LD HL,BCDADR	
C70	77		LD (HL), A	
C71	23		INC HL	
C72	77		LD (HL), A	
C73	23		INC HL	
C74	77		LD (HL), A	
C75	4A		LD C, D	
C76	CD 53 0C		CALL SOT	
C79	4B		LD C, E	
C7A	CD 53 0C		CALL SOT	
C7D	C9		RET	
	; SEGUE PROGRAMMINO DI PROVA:			
C7E	00 00	DATOBIN	DEFW 0	
C80	21 7E 0C		LD HL, DATOBIN	
C83	56		LD D, (HL)	
C84	23		INC HL	
C85	5E		LD E, (HL)	
C86	CD 6C 0C		CALL BCDCONV	
C89	76		HALT	
			END	

Altre conversioni

A) *Da BCD ad ASCII*

Come già detto, le cifre decimali da 0 a 9 sono espresse, in esadecimale, con i codici da 30H a 39H. Nel caso dell'EBCDIC (o BCD "esteso") ogni byte contiene una sola cifra decimale, con il semibyte di sinistra riempito di "uni" (valore esadecimale F), in funzione di zonatura. In questo caso il pro-

blema è banale, bastando la sostituzione della zonatura F con la 3.

Supponendo ad es. in B la cifra da convertire, basterà fare:

LD A, B poi AND A, 3FH

L'AND col dato immediato 3FH lascia immutati i bit di destra (dato che il prodotto logico $1 \cdot x = x$ per qualsiasi valore del bit x) e, nel semibyte di sinistra l'AND tra 1111 e 0011 dà 0011B = 3H.

Nel caso invece di cifre BCD "impaccate" - il che vuol dire due cifre per byte, per fissare le idee consideriamo sempre il registro B - le cose sono meno agevoli. Occorre *"disimpaccare"* ed aggiungere la zonatura 3 (si fa rilevare che il problema è quasi il medesimo che il disimpaccamento in BCD, tranne che la zonatura da aggiungere è F anziché 3).

Sia DE un puntatore alle posizioni di memoria sulle quali si vuole caricare la coppia di byte convertiti. Qui ci converrà, per visualizzare in modo immediato il risultato (e la correttezza della routine), scegliere posizioni di memoria della video RAM del nostro sistemino di sviluppo.

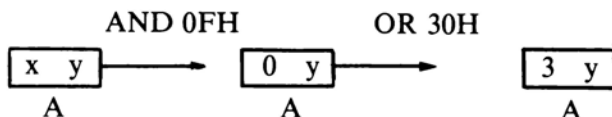
Una prima soluzione, terminata dalla istruzione RET tipica di una subroutine, potrebbe essere:

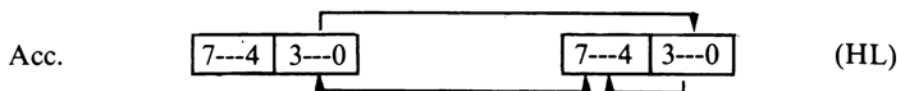
```
LD      A,B
RRCA
RRCA
RRCA
RRCA
AND     0FH
OR      30H
LD      (DE),A
INC     DE
LD      A,B
AND     0FH
OR      30H
LD      (DE),A
RET
```

Dopo il caricamento di B in A, con quattro rotazioni circolari (destre o sinistre, non importa) si ottiene l'inversione dei semibyte. Detti x e y i contenuti di ciascuno di essi, la cosa è illustrata qui di seguito (con le frecce che stanno ad indicare successive trasformazioni, non solo caricamenti):

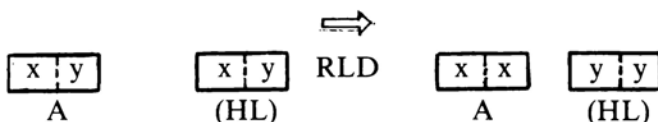
A questo punto, le due successive operazioni logiche portano, la prima, zero al posto di y, poi la zonatura 3 nel semibyte sinistro:

Si pone A in (DE) poi, dopo esser passati, con INC DE, alla posizione di memoria seguente, si ricarica B in A e si ripetono le due operazioni logiche,

stavolta col semibyte y:

La routine termina col caricamento in (DE).

Una soluzione altenativa si può ottenere con l'istruzione, tipica del nostro Z80, RLD (Rotate Left Decimal, o anche la RRD) di cui qui riportiamo per comodità la rappresentazione grafica del modo di operare:

Ovviamente, ora *è d'obbligo* l'uso del puntatore HL. Si ha:

LD (HL),B poi LD A,B e RLD ecc.

Il contenuto di B è caricato, oltre che in A, anche in (HL). Dopo l'esecuzione di RLD la situazione è:

A questo punto, ci conviene fare così:

DEC HL	ci spostiamo sul byte di sinistra;
AND 0FH poi OR 30H:	poniamo in A 3 X;
LD (HL),A	poniamo 3 X nel primo byte di memoria;
INC HL	passiamo al byte accanto, dove, ricordiamo, *c'è ancora* yy;
LD A,(HL)	poniamo yy in Accumulatore;
AND 0FH poi OR 30H:	si fa A = 3y
LD (HL),A	si carica 3y nel byte di destra.
RET	

Totale istruzioni: 18 byte, contro ... altrettanti del caso precedente.

B) *HEX convertito in rappresentazione ASCII*

Può essere utile, nel caso del nostro NASCOM I, una routine di conversione volta a rappresentare sullo schermo TV il contenuto esadecimale di un byte *qualunque ne possa essere il significato.* In questo caso, ogni semibyte può contenere anche le *cifre hex.:* A, B, C,..., F. Se vogliamo rappresenta-

re con queste medesime lettere le configurazioni binarie da 1010 a 1111, tenendo presente che il circuito integrato ROM generatore di caratteri ha bisogno dei codici 41H, 42H, 43H, ..., 46H rispettivamente, si vede allora che adesso è necessario, dopo aver portato nel semibyte di destra di A la cifra da convertire, aggiungere 37H, cioè 00110111B.

Se infatti ad esempio nell'Accumulatore c'è 0CH = 00001100B, sommandovi 37H si ottiene:

Hex.		bin.	
	0C +		0000 1100 +
	37		0011 0111
	43		0100 0011

Quando invece la cifra è minore o uguale a 9, si deve aggiungere 30H (o, che è lo stesso agli effetti pratici, fare l'OR con l'immediato 30H, come fatto nell'esempio che precede).

Anche ora vogliamo supporre che il dato di partenza si trovi in B.

Il programmino è riportato qui di seguito.

Programma P. 5-4

ROUTINE CRTHEX PER DATI HEX SU SCHERMO

Registro B= Dato; *coppia DE* = puntatore

Indir.	Contenuto	Label	Assembly	Commenti
			ORG D00H	
D00	08	CRTHEX	EX AF,AF'	salva Acc. & flag
D01	CD 09 0D		CALL GIRL	conversione 1° semibyte
D04	CD 09 0D		CALL GIRL	conversione 2° semibyte
D07	08		EX AF,AF'	repechage di Acc. & flag
D08	C9		RET	fine CRTHEX
D09	78	GIRL	LD A,B	
D0A	07		RLCA	rotazione a sinistra
D0B	07		RLCA	di
D0C	07		RLCA	quattro
D0D	07		RLCA	bit (°)
D0E	47		LD B,A	
D0F	E6 0F		AND 0FH	azzera semibyte sin.
D11	FE 0A		CP 10	cfr. con 10 = AH
D13	D2 1A 0D		JP NC,GIU	se = A, B,..., F vai GIU'
D16	C6 30		ADD A, 30H	se = 0, 1,..., 9 agg. 30H

(°) - Si noti che fare quattro RLC B direttamente sul registro B, a parte ogni altra ovvia considerazione sul prosieguo del programma, *non conveniva* comunque, richiedendo 4 × 2 = 8 byte contro i 4 + 2 di sopra.

Indir.	Contenuto	Label	Codice Assembly	Commenti
D18	18 02		JR PIUGIU	poi, vai ancor PIUGIU'
D1A	C6 37	GIU	ADD A, 37H	se>9 aggiungi 37H
D1C	12	PIUGIU	LD (DE),A	dato rappr. ASCII in TV
D1D	13		INC DE	prepara posiz. accanto
D1E	C9		RET	fine GIRL.
	; SEGUE PROGRAMMETTO DI PROVA:			
D1F	11 4A 09	PROVA	LD DE, 094AH	
D22	3E 1E		LD A, 1E	istruzioni per
D24	CD 3B 01		CALL 13BH	pulire TV (v. cap. 3)
D27	06 F0		LD B,F0H	
D29	CD 00 0D	RIPETI	CALL CRTHEX	
D2C	04		INC B	
D2D	28 03		JR Z,FINE	
D2F	13		INC DE	(spazio di separazione)
D30	18 F7		JR RIPETI	
D32	76		HALT	(oppure le solite istruzioni
			END	di ritorno al monitor).

Gli si è dato il nome di CRTHEX. Vediamolo in dettaglio. L'inizio con la istruzione EX AF,AF, serve all'(eventuale) salvataggio dei dati contenuti nella coppia AF. Istruzioni di questo genere (o anche dei PUSH) sono tipiche, all'inizio e alla fine, nelle routine di utilità; si evita in tal modo la necessità di compiere simili operazioni nel programma chiamante (risparmiando anche qualche byte, se la routine è chiamata molte volte).

In altri casi si ricorre alla potente EXX che scambia, con i registri "fantasma" corrispondenti le coppie BC, DE ed HL. Si osservi che, nel nostro esempio non è il caso di salvare tali coppie perchè la routine non usa HL, nonchè il registro C, mentre B e DE sono impiegati con il medesimo scopo sia da CRTHEX che dal chiamante (v. programma PROVA).

Segue una duplice CALL GIRL (sia perdonata la piacevolezza ...), ove GIRL è a sua volta una routine che ha la funzione di convertire successivamente e trasferire in (DE), cioè per quanto detto sullo schermo, di due semibyte di B. Vediamo pertanto come lavora questa ... ragazza.

- LD A,B seguita da 4 volte RLCA, poi LD B,A: queste prime istruzioni hanno lo scopo di invertire, in A, le cifre x e y; quindi A è ricaricato in B. In tal modo, alla seconda chiamata di GIRL, la ripetizione delle quattro RLCA ecc. mette in A, poi anche in B, le cifre x e y *non rovesciate:*

Acc.	*B*	
- -	x y	(situazione prima di LD A,B fino LD B,A alla prima CALL GIRL);
y x	y x	(situazione dopo);
x y	x y	(dopo seconda CALL GIRL, fino a LD B,A)

- AND 0FH e seguenti: il resto di GIRL consiste nell'azzeramento del semibyte sinistro, poi il confronto del risultato (o x oppure o y) con 10 (o meglio AH): se x (oppure y, alla seconda chiamata) è maggiore di 9 (N.B. non si dimentichi che la condizione NC significa maggiore o uguale!) si corregge con ADD A,37H, altrimenti con ADD A,30H. Segue il caricamento in (DE) e l'incremento di DE, per predisporre il puntatore all'indirizzo successivo.

Programma MAIN di collaudo (PROVA)

Lo scopo, qui è di far apparire - distanziati tra di loro di un blank - coppie di caratteri da F0H a FFH (°).

Le istruzioni di PROVA sono scritte subito dopo quelle della subroutine CRTHEX. Il comando "E" dell'Execute va quindi dato con l'indirizzo D1F.

Si parte con DE = 094A (indirizzo primo byte riga tivù omonima) e con B = F0H, dopo le istruzioni LD A,1EH seguita da CALL 13BH (ossia CRT del monitor) che, ricordiamo, attuano il "clear" dello schermo (dovremmo in teoria metterle in gioco in tutti gli inizi di programmi che utilizzano lo schermo, onde evitare di doverlo fare manualmente ...).

Chiamando CRTHEX, la coppia F0 viene scritta sullo schermo, posizioni 094A e 094B. Quindi si accresce di un'unità B e se non è B = 0 si incrementa DE, il che serve a saltare una cella della riga, a titolo di spaziatura, poi si ripete il ciclo (RIPETI). Quando B arriva al valore FF, il successivo INC B dà B = 00 ed il controllo passa a FINE, dove si ha l'HALT. (°°)

Non è difficile rendersi conto che l'intera riga di 48 posizioni di memoria è riempita: essendo tre (2 cifre + 1 blank) i byte impegnati per ogni elemento Fx (x = 0, 1, 2, ... , F), si ha 3 × 16 = 48.

Questo esempio di test mostra chiaramente come sia possibile escogitare sistemi di "debug" più rapidi e diretti che non quelli usuali del "breakpoint" o del "single step". Con un po' di inventiva, lo schermo TV è utile a questi fini nella maggior parte dei casi.

Vediamo ora una variante. Togliamo di mezzo le due EX AF,AF, che in fondo costituiscono un pò una finezza. Anzichè chiamare due volte GIRL si potrebbe fare una sola chiamata, quindi togliere la seconda CALL GIRL

(°) - Chi legge può ovviamente fare la prova con caratteri la cui prima cifra sia diversa da F o di impiegare ogni programma di collaudo più semplice o complesso (es. far scrivere, su 15 righe TV, i caratteri, in ordine, da 11 a FF).

(°°) - Abbiamo qui un trucchetto alternativo rispetto al solito de-contatore per numerare dei passi *in avanti* in quantità pari a 16 o 32, fino a 256.

e la RET. Il programma in definitiva si riduce a:

```
CRTHEX      CALL GIRL
GIRL        LD A,B ecc. fino a:
            .........
PIUGIU      LD (DE), A
            INC DE
            RET
PROVA       .......
```

ed il marghingegno "gira" così: con la CALL GIRL si esegue una prima volta GIRL, quindi si va alla successiva istruzione, cioè a ... GIRL. Questa, come è giusto, è eseguita una seconda volta poi la RET all'indirizzo D1E fa tornare alla istruzione *successiva* a quella del MAIN che ha chiamato CRTHEX.

Insomma, questa RET opera, la prima volta, come istruzione terminale della GIRL, la seconda come rientro da CRTHEX. Si risparmiano così 4 byte: 3 di una CALL GIRL più una RET. Metodo interessante, ma un pò ... funambolico, cui preferiamo pertanto il primo, per motivi di chiarezza.

Un sottoprogramma di questo tipo e con queste medesime funzioni è, come dovrebbe essere evidente, incorporato nel monitor NASBUG (serve quando si deve eseguire il comando M, per dare la rappresentazione sullo schermo del contenuto Hex. di un byte di memoria): esso rassomiglia abbastanza al nostro (ma giuriamo di non averlo copiato).

Programma P. 5-5

CONVERSIONE ASCII (tastiera) HEXADECIMAL

È la conversione opposta a quella vista in precedenza. È utile quando si vuole introdurre la tastiera un dato esadecimale *nel corso di un programma.*

La routine KBD, da sola, non serve allo scopo, in quanto il corrispettivo del tasto premuto è in ASCII, nell'Accumulatore. Viceversa il comando "M" opera in modo semplice nella fase di caricamento dati hex. in memoria e "debug" (sfruttando una routine del tipo di quella che stiamo per illustrare) ma non può evidentemente essere invocato nel corso della esecuzione di un programma.

La routine che qui si propone è, vedremo, abbastanza semplice.

Un dato esadecimale deve essere introdotto con due battute di tasti variabili da 0 fino a 9 poi A, B, ..., F(ASCII: 30 ÷ 39 e 41 ÷ 46). La correzione da effettuare è *opposta a quella del caso precedente:* sottrarre, anzichè sommare 30H e 37H nei due casi. Vediamo la codifica.

Programma P.5-5

Indir.	Contenuto	Label	Cod. Assembly	Commenti
			ORG C50H	
C50	CD 69 00	KBDRIP	CALL 69H	chiama routine KBD
C53	30 FB		JR NC, −3	attendi un carattere
C55	FE 40		CP 40H	maggiore di 40H? (°)
C57	30 F8		JR NC, SOTTO	se SI, vai SOTTO
C59	D6 30		SUB 30H	se NO, togli 30H e
C5B	18 02		JR RISOTTO	va a RISOTTO (s.rel. +4)
C5D	D6 37	SOTTO	SUB 37H	
C5F	CB79	RISOTTO	BIT 7,C	come è il bit 7 di C?
C61	20 08		JR NZ,GIU	se è = 1,va GIU
C63	CB F9		SET 7,C	se = 0 ponilo a 1, poi
C65	0F		RRCA	esegui
C66	0F		RRCA	quattro
C67	0F		RRCA	rotazioni
C68	0F		RRCA	di bit (0xdiventa x0)
C69	47		LD B,A	risultato in deposito
C6A	C9		RET	
C6B	B0	GIU	OR B	alla 2ª chiam.: x0+0y=xy
C6C	C9		RET	
C6D	CB B9	KBDHEX	RES 7,C	reset switch (inizio rout.)
C6F	CD 50 0C		CALL KBDRIP	
C72	CD 50 0C		CALL KBDRIP	
C75	C9		RET	fine rout. principale.

La routine principale, KBDHEX, consiste in due chiamate consecutive di una subroutine più interna detta KBDRIP, con la prima chiamata preceduta dal reset del bit 7 del registro C. Questo bit serve da *switch,* in modo che, nelle due successive esecuzioni di KBDRIP, questa operi in due modi leggermente diversi. Vediamo come.

Routine KBDRIP

La prima volta che essa è evocata, la ormai ben nota chiamata della routine KBD del monitor seguita dall'usuale salto relativo di −3, fa attendere la battuta di un carattere. Quando questa avviene, con Cy = 1, si va al confronto con 40H e, se non si supera tale valore, si opera la correzione di −30 con SUB 30H; poi si fa JR RISOTTO (altra innocente plaisenterie). Se invece l'Accumulatore supera 40H, cioè si tratta dell'ASCII corrispondente ad A, B ecc., dopo il salto a SOTTO, la correzione è di −37H.

(°) - È sottinteso che si battano solo i tasti 0 ... 9 e A ... F

A questo punto, alla label RISOTTO, si fa il test dello switch: esso alla prima chiamata di KBDRIP è "off", quindi non si va "GIU" e, dopo aver fatto il set del bit 7 (si perdoni la cacofonia) di C, si eseguono quattro rotazioni circolari di A. Queste fan sì che, detto x l'esadecimale corrispondente alla battuta fatta, l'Accumulatore contiene ora x0. È un dato ancora incompleto che comunque è messo in deposito in B.

Senza star a rifare i disegnini degli esempi precedenti, facciamo il caso che venga battuto il tasto C. Nell'accumulatore c'è inizialmente 43H.
Dopo che gli è stato tolto 37H, A diventa, come è giusto, 0CH. Con quattro RLCA si ha A = C0H, idem B, dopo LD B,A.

Alla seconda chiamata di KBDRIP (in cui viene attesa e poi ricevuta la seconda cifra y), arrivando al ... RISOTTO, cioè dopo la solita correzione di − 30H o − 37H, il test del deviatore fa saltare stavolta alla label GIU, ove si effettua l'OR con il registro B. (°). Il risultato è facile da capire:

a questo punto in A c'è 0y, mentre in B è, si è visto, x0, pertanto l'OR detto produce A = xy, cioè la coppia di cifre esadecimali che si sono battute, in due tempi, da tastiera.

Segue un RET che è quello definitivo: il risultato si trova nell'Accumulatore.

A questo punto, lasciamo al lettore l'onere di farsi, in analogia con gli esempi che precedono, un programmino opportuno di prova.

A chiusura di queste ultime subroutine di conversione, proponiamo invece un lavoro di sintesi e ripasso.

Esercizio - Creare un programma che operi la moltiplicazione (binaria) tra dati esadecimali a 16 bit, utilizzando la routine MOLT16 (sottoprogramma P. 3-4 del capitolo 3), con i valori del MDO e MRE *introdotti da tastiera.*

In altri termini, si vuole ottenere che la introduzione dei dati avvenga durante e non prima del programma (prevenendo così la, sia pur troppo facile, derisione degli sprovveduti che, quando ci vedono fare la prova di un programma del genere ci accusano di aver fatto un programma per ... moltiplicare sempre 3 per 2!).

Immaginiamo di riscrivere (riassemblando con un compilatore o ... a mano) le varie subroutine che ci servono , in testa (i puntini sottintendono le già viste istruzioni dei vari "sorgente"). Si ricordi poi anche che:

(°) - Altrettanto bene andava l'istruzione ADD A,B.

- la nostra CRTHEX ha bisogno del dato da inviare sullo schermo nel registro B e del posizionamento del puntatore DE alla cella video che si vuole (es. la 0BD0 della "top row");
- la MOLT16 ha inzialmente MDO in HL; MRE in DE, poi il risultato PR è, al termine, in HL.

Con queste premesse, non è difficile ottenere (in solo simbolico):

Label	Codice Assembly	Commenti
	ORG	
CRTHEX	EX AF,AF'	
	CALL GIRL ecc.	
....	....	
KBDRIP	CALL 69H	(inizio "fisico" di KBDHEX)
....	....	
KBDHEX	RES 7,C	
	CALL KBDRIP ecc.	
....	....	
MOLTI6	LD B,16 ecc.	
....	....	
KBDMOLT	CALL KBDHEX	prime due
	LD H,A	cifre in H
	CALL KBDHEX	altre due
	LD L,A	cifre in L (fine MDO)
; segue ora il MRE:		
	CALL KBDHEX	
	LD D,A	prime due cifre in D
	CALL KBDHEX	
	LD E,A	altre due in E (MRE)
	CALL MOLT16	moltiplica
	LD B,H	posiz. dato in B
	LD DE, OBDOH	posiz. DE (top row)
	CALL CRTHEX	scrivi prime due cifre PR
	LD B,L	"low order" PR in B
	CALL CRTHEX	scrivi altre cifre PR
	....	RESTO DEL PROGRAMMA

Da quest'ultimo esempio non si deve pensare che la vita sia sempre così difficile. C'è infatti da tener conto che: a) il nostro personal computer ha un hardware predisposto per operare come "debugger" e non come calcolatore; b) ad ogni buon conto, nel suo monitor le routine di conversione *ci sono già;* c) in un sistema orientato al calcolo, in genere nel firmware sono predisposte routine di calcolo abbastanza complesse, come la divisione e anche tavole di funzioni matematiche comuni ecc.

Cionondimeno, deve risultare abbastanza chiaro che in questi casi, specialmente con un microprocessore a 8 bit, *è indispensabile l'uso di linguaggi* di alto livello (come il FORTRAIV, il PL/Z o almeno il BASIC).

CAPITOLO 6

SIMULAZIONE DI IMPIANTI E CONGEGNI: SEMAFORO E OROLOGIO-CRONOMETRO

In questo come in capitoli successivi, vedremo degli esempi di simulazione di impianti o dispositivi in genere a microprocessore.

Dopo l'infernale capitolo precedente pensiamo che il lettore debba ora essere stimolato con più interessanti esperienze ad accostarsi alla materia con simpatia. Beninteso nel capitolo precedente si trattava di programmini indispensabili per comprendere a fondo il software, ma, grazie a dio, il microprocessore trova il suo campo più tipico in questo settore, dove i problemi di tipo calcolistico sono in genere meno preponderanti, ma rimangono, anzi per certi versi si acuiscono, quelli di tipo *logico.*

Si dice anzi che l'obiettivo preminente che è stato alla base della progettazione e dello sviluppo dei micro sia stato appunto quello di sostituire i *controller,* tradizionalmente implementati mediante *circuiti logici* (la cosiddetta logica cablata).

Nelle applicazioni che stiamo per vedere, i circuiti tradizionali prevedono l'uso di contatori (quando non addirittura temporizzatori di tipo analogico) e porte tipo "and", "or" ecc. Noi realizziamo il tutto con del software, almeno in prevalenza.

Simulazione significa per noi utilizzare i semplici mezzi di I/O del nostro NASCOM - la tastiera e l'unità video - per imitare con la prima comandi e segnali provenienti dall'esterno e, con la seconda, segnali di output di ogni genere.

Per essere precisi, nel primo caso, dato che la keyboard è connessa a dei porti di I/O la simulazione è nel complesso abbastanza fedele. Con l'unità video, che nel nostro caso visualizza dati esistenti su di una memoria RAM, almeno a prima vista, non si può dire altrettanto. A questo proposito va però detto:

1) le operazioni di output tramite un porto di I/O e quella di scrittura in una RAM sono concettualmente apparentate, nel senso che sempre di informazioni provenienti dal microprocessore si tratta;
2) la semplice, ma abbastanza frequente tecnica del "memory mapping", di cui parleremo più diffusamente nel seguito, è, anche formalmente, identica.

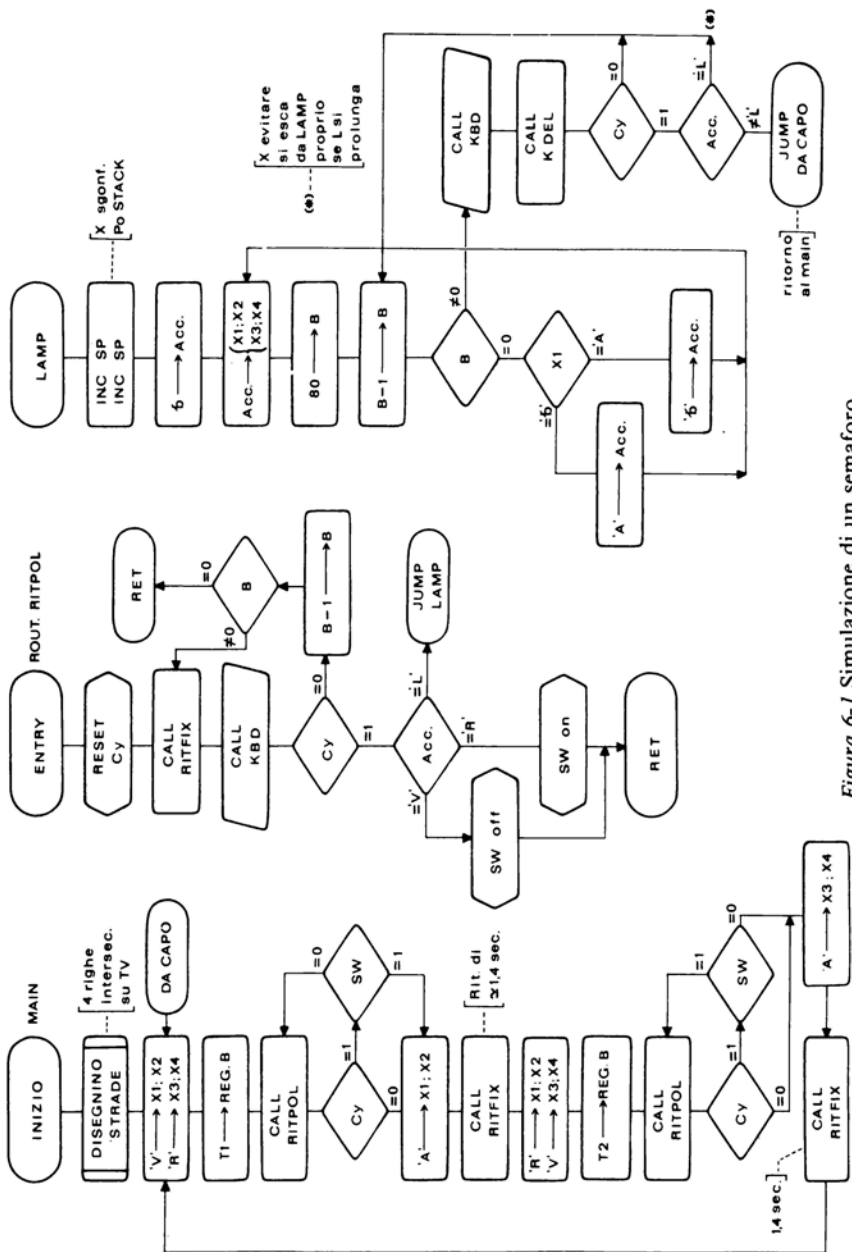



Figura 6-1 Simulazione di un semaforo.

Programma P. 6-1 SIMULAZIONE DI UN SEMAFORO

Obiettivo. Si vuol rappresentare, utilizzando come si è detto la tastiera e lo schermo TV del NASCOM, il funzionamento di un semaforo, operante su un normale quadrivio.

Dalla tastiera si possono battere i tasti "V", "R" o "L". Quando nessun tasto è premuto, il semaforo, visualizzato, come vedremo meglio, con delle lettere al centro del contorno delle quattro strade viste dall'alto, sullo schermo, evolve automaticamente dal verde all'arancio al rosso, su ciascuna direzione di marcia. Quando si batte "L", si passa al funzionamento *lampeggiante.* Con il tasto "V" si chiede il verde sulla strada rappresentata verticalmente sul video, con "R" si chiede il rosso su questa medesima direzione, come dire il verde sull'altra.

Se la richiesta è difforme dallo stato presente del semaforo, si effettua la commutazione anticipata allo stato successivo (tramite naturalmente il normale stato intermedio di arancio prima del rosso). Se invece essa suona di conferma per lo stato presente (es. richiesta di verde sulla via verticale, quando su questa direzione di marcia il verde c'è già) si resta nello stato medesimo, *senza che ciò comporti alcun suo prolungamento.*

Questa impostazione, che naturalmente non è la sola possibile, ci sembra la più semplice e meno pericolosa. Si può infatti pensare che i tasti, oltre che essere azionati manualmente (da un vigile dalle braccia stanche) siano collegabili ad una rete logica esterna che, sulla base di opportuni sensori posti sulle sedi stradali, tenga conto delle condizioni di maggior traffico. Detti T1 e T2 i tempi assegnati a ciascuna direzione, se la riconferma di verde su una di esse dovesse significare un suo prolungamento per ulteriore tempo T1, si potrebbe avere il rischio di un monopolio di verde da parte di questa, in una situazione di traffico molto intenso.

Naturalmente una impostazione come questa adottata è, peraltro, semplificata. Infatti un comando "V" o "R" troppo prolungato significherebbe finestre di verde troppo esigue come durata per l'altra strada. La cosa migliore sarebbe quindi quella di prevedere anche dei tempi minimi maggiori.

Questo però complicherebbe più del necessario l'esempio didattico.

Flow chart

Esso è disegnato nella figura 6-1. Sulla sinistra è rappresentato il MAIN program. Si inizia con il sottoprogramma DISEGNINO STRADE nel corso del quale si fanno apparire sullo schermo video quattro linee intersecantesi (°). Al centro di esse, nelle quattro posizioni indicate simbolicamente

---

(°) - I perfezionisti saranno costretti dalla loro pignoleria a prevedere dei blank nella parte centrale, in modo che appaiano bene gli angoli.
Ciò però richiede altre, noiose, istruzioni.

con X1 e X2 (indirizzi della video RAM 0962H e 09E2H), nonchè X3 e X4 (indirizzi 09A1 e 09A3) posti, i primi, uno sull'altro, mentre gli altri due sono in orizzontale, si andranno a rappresentare con delle lettere: **V** per Verde, **R** per Rosso ed **A** per Arancio, la condizione delle facce semaforiche volte verso ciascuna delle quattro vie. Dato che ci si riferisce al caso più banale, X1 e X2, come pure X3 e X4, hanno sempre il medesimo valore nello stesso momento. Una impostazione di questo tipo, peraltro, oltre che servire a dare un'idea visiva migliore, ci sembra si presti meglio ad una generalizzazione ad altri casi.

Dopo il disegnino delle strade, si carica "V" in X1, X2 e "R" in X3, X4.

Comparirà quindi una figura come la seguente, il cui significato è di immediata evidenza (verde sulla direzione verticale - strada "stretta", rosso su quella orizzontale - via "larga"):

```
         l         l
         l         l
  ___________________________
         l         l
         l    V    l
         l   R R   l
         l    V    l
         l         l
  ___________________________
         l         l
```

A questo punto, posto T1 nel registro B, si esegue la routine RITPOL (che vuol significare RITardo più POLling, cioè esplorazione della tastiera).

Da questa routine si esce dopo un tempo corrispondente a T1 volte il ritardo fisso di circa 1,4 secondi della routine RITFIX, qualora non si premano tasti, altrimenti si esce anticipatamente. Nel primo caso, il flag Cy è off, onde si prosegue con il funzionamento automatico. Se, invece, il Carry flag è acceso, il passo successivo dipende dal deviatore SW. Se tale switch è uguale a zero, come si vedrà analizzando in dettaglio la RITPOL, ciò significa che è stato premuto il tasto "V". Come dire che questa richiesta - magari per un errore - suona di conferma alla situazione attuale: pertanto si torna a chiamare la subroutine RITPOL, però *senza ripristinare T1* in B. In tal modo questo registro completa, in RITPOL, il suo ciclo di de-contatore. I motivi di ciò sono già stati anticipati, perciò non insistiamo.

Con SW = 1, che implica tasto battuto = "R", si deve invece anticipare la commutazione da verde a rosso sulla strada "stretta" (viceversa su quella "larga"). In tal caso, come pure naturalmente quando qui si giunge con Cy

= 1, innanzitutto si pone "A" in X1 e X2, visualizzando la figura sotto, che resta per i circa 1,4 secondi della routine di ritardo RITFIX, la quale, come si comprende, è stata scelta come unità base di tempo, onde T1 e così pure T2 sono fattori moltiplicativi del tempo dell'arancio, identico in entrambe le direzioni.

Si passa quindi a caricare "R" in X1 e X2, mentre "V" è posto nelle posizioni X3 e X4, si trasferisce T2 in B e si evoca nuovamente la RITPOL. Tutto funziona come prima, tranne un piccolo particolare: se Cy = 1, cioè carattere premuto, la commutazione di stato è anticipata (ossia si dà nuovamente verde sulla direzione verticale) con il deviatoio SW = 0 anzichè, come prima, con SW = 1, mentre con SW = 1, si chiama ancora la RITPOL. Il motivo è più che evidente: stavolta è SW = 0, cioè richiesta, di verde sulla via verticale, a determinare il cambio di stato. Questo, ora, si effettua con l'immissione di una coppia di "A" sulle posizioni X3 e X4, per la solita durata di 1,4 sec., poi si riprende alla label DACAPO, ovverosia si riprende il ciclo.

Routine RITPOL

Si tratta di un sottoprogramma abbastanza interessante, in quanto ad uscite multiple. Anzi, come si vedrà, una di queste uscite, quella corrispondente alla battuta della chiave "L", anzichè terminare con una RET, finisce con un salto diretto ad un punto del MAIN diverso da quello in cui la RITPOL era stata chiamata. Questa piccola audacia va usata con la dovuta cautela, ad evitare rischi seri. Vedremo come.

Si comincia con il reset del Carry. Questa istruzione in realtà è superflua, dal momento che la routine KBD del monitor esce sempre con Cy = 0 se nessun tasto è premuto. Si tratta quindi di un pleonasmo, la cui origine, psicologica, è legata ad un dubbio sorto in fase di codifica. L'abbiamo la-

sciata ugualmente, sia per non dover riassemblare il tutto per una simile inezia, sia perchè, in fondo, questo fatto è abbastanza istruttivo.

In casi consimili infatti, può essere buona norma, quando si hanno dubbi sul reale modo di operare di un sottoprogramma (e non si ha tempo di andarlo a rivedere) mettere istruzioni-catenaccio del genere (in questo caso, lo scopo era di far sì che, al rientro in RITPOL, dopo esserne in precedenza usciti con Cy = 1, il Carry venisse prudenzialmente subito rimesso a zero): un pò di pessimismo non guasta. E in certi casi, si evitano errori.

Se poi, in fase di revisione, ci si accorge che certe istruzioni sono superflue, le si può sempre lasciare o anche sostituire con dei NOP (No OPeratio, codice assoluto 00).

Dopo questa digressione forse un pò più lunga del necessario, vediamo come marcia questa RITPOL.

Si fa trascorrere un tempo dato dalla solita RITFIX, quindi si esegue la CALL di KBD e se da essa si esce con Cy spento, si decrementa il registro B, ricominciando il loop con RITFIX, fino a che non è B = 0 e in tal caso si esce da RITPOL dopo un lasso di tempo pari a T1 o T2 volte 1,4 secondi.

Qui, come pure a proposito del gioco del deviatore SW, è d'obbligo la considerazione seguente: se non avessimo escogitato un sottoprogramma un po' flessibile come così, come è stato fatto, risulta RITPOL, avremmo dovuto scrivere due volte gruppi di istruzioni che differiscono soltanto in modesti particolari.

Un'altra osservazione si riferisce al fatto che, a ben guardare, il ritardo fisso di 1,4 sec. è un pò lungo. Ciò comporta la necessità di *tenere ben pigiati i tasti*; se ciò non avviene, la KBD rischia di non operare, se durante RITFIX il tasto è rilasciato! È stata fatta questa scelta allo scopo di ottenere un programma più semplice (e compatto).

Il lettore provi a modificare il programma: occorre un minore ritardo (per esempio 100 msec., o anche abolire la chiamata di RITFIX entro la RITPOL, affidandosi ai soli 1,5 msec. implicati dalla KBD. In quest'ultimo caso, per avere dei tempi ragionevoli di verde e/o rosso, sarà necessario caricare T1 o T2 nel registro doppio BC, anzichè in B ecc. ecc.

Peraltro non è escluso che anche in un'applicazione reale non sia proprio richiesto che un certo comando duri così a lungo per essere accettato.

Quando un tasto è premuto, cioè Cy = 1, il passo seguente dipende da quale chiave è stata battuta: se l'Accumulatore, in cui la KBD pone il carattere, in ASCII code, contiene "V" viene spento lo SWitch (come si potrà vedere, esso è costituito da un bit del registro C), mentre SW è posto in ON se si ha a che fare con "R". In tal modo, tornando al punto di chiamata da parte del MAIN, si ha l'informazione relativa alle condizioni che nella subrou-

tine si sono manifestate: Cy = 0 o = 1 indica se vi è stata o meno una battuta di tasto; mentre SW con il suo stato ON o OFF segnala quale dei tasti "V" o "R" è stato pigiato.

A dire il vero, si sarebbe potuto anche fare a meno di SW. Infatti l' Accumulatore, ovviamente, conserva il contenuto precedente l'uscita dalla RITPOL e, quindi, il test di SW poteva benissimo essere sostituito dalla comparazione del registro A con l'immediato "V" o "R", mentre entro la routine RITPOL ci si poteva limitare a confrontare A con "L" solamente, senza poi preoccuparsi della messa in "on" o "off" di SW.

Padronissimi, per questo, di raccomandarci, per una assunzione, all'U.C.A.S. (Ufficio Complicazioni Affari Semplici), ma il fatto è che questa impostazione ci è apparsa più facilmente generalizzabile (per es. al caso in cui, anzichè con una tastiera, si ha a che fare con semplici connessioni ad un porto di I/O, senza memorizzazione del dato ecc.).

Anzichè usare il registro C per il solo bit di SW si sarebbe potuto anche impiegare uno dei bit 7 o 6 dell'*Accumulatore*. Infatti tutti i caratteri ASCII hanno tali due bit nulli e comunque la rilevazione del tipo di carattere battuto *non interessa* il MAIN. L'impostazione prescelta ci è però parsa meno cervellotica (e di registri, in questo problemino, ce n'è a iosa).

A questo punto, resta da vedere che cosa accade quando si batte "L".

Si va al connettore denominato con la label LAMP, dove iniziano istruzioni che servono a simulare semaforo lampeggiante, ossia alternativamente emettente "arancio" (carattere "A") e "buio" (blank), su tutte e quattro le X. Questo ramo di sottoprogramma termina con un salto alla label DACAPO del MAIN (subito dopo il sottoprogramma DISEGNINO), anzichè con il consueto RET. Ciò può apparire poco ortodosso, ma si pensi a che cosa sarebbe successo se avessimo fatto l'uscita con RET: supponendo che fosse stato prescritto rigorosamente di proseguire con DACAPO (°). Avremmo dovuto aggiungere un ulteriore test a quelli di Cy e di SW, all'uscita di ciascuna delle due CALL di RITPOL. Ora è certo più semplice il "jump" diretto a DACAPO, anche per la chiarezza. Ma che cosa accade, se da una routine *si esce senza la RET?* Si ricordi che dopo la chiamata di una routine, la CPU automaticamente carica sullo stack i due byte dell'indirizzo dell'istruzione successiva a quella in cui viene fatta la CALL. Se ora si fa il

---

(°) - Questa prescrizione è quasi certamente poco realistica nel caso presente (si può quasi sicuramente accettare il rientro da LAMP "dove capita"). Il reale motivo dell'impostazione vista è didattico: si vogliono indicare trucchi di programmazione che possono tornare utili in altri casi (lo vedremo tra pochissimo, nell'esempio dell'orologio).

salto a DACAPO, sul PC (Program Counter) viene caricato l'indirizzo giusto, quello della istruzione etichettata con la label DACAPO e l'indirizzo caricato sulla "catasta" (stack), non ci occorre più, quindi nulla di male da questo punto di vista. *Ma il guaio sta nel fatto che, ogni volta che si va a LAMP, lo stack si "gonfia"* sempre di più di tali indirizzi inutili. C'è il rischio che, per quanto corto sia il nostro programmino e per quanto lontana sia la zona riservata alla catasta questa alla fine invada aree di memoria riservate al programma! Come si è già detto nel secondo capitolo, se il programma è su RAM, come avviene con la nostra simulazione, ne vengono modificate le istruzioni, mentre, se si tratta di un "controller", il programma è su ROM e non viene modificato, ma lo stack va in "*overflow*", ossia non ha più spazio per cumulare indirizzi di chiamata. Lasciamo decidere a chi legge quale dei due inconvenienti dia il risultato più disastroso.

Ma il rimedio c'è ed è semplice: si tratta di scrivere esplicitamente istruzioni di "sgonfiamento" dello stack (che invece sono implicate dalla RET).

Ciò è stato fatto ponendo, all'inizio stesso di LAMP due istruzioni INC SP (per non dimenticarsene! ovvio che andavano bene anche alla fine, prima di JP DACAPO). Queste due INC SP assolvono il compito posizionando il puntatore SP dello stack due byte più avanti.

Il resto è abbastanza semplice. Dopo aver caricato dei blank in X1, X2, X3 e X4 e 80 in B (allo scopo di creare, per il buio e l'arancio, una durata di mezzo secondo, più o meno). Si decrementa il decontatore e, se diverso da zero e dopo un ritardo di circa 6-7 msec. (5 msec. di KDEL più 1,5 msec. della KBD) si ritorna a decrementare B se la chiamata di KBD dà Cy = 0.

Con Cy = 1, si fa il test dell'Accumulatore e si torna a diminuire di uno il registro B, se il tasto battuto è "L". Ciò soprattutto per evitare che si esca da LAMP proprio se il tasto "L" è premuto un po' a lungo!

Con tasti diversi da "L" si ritorna al punto del MAIN etichettato DACAPO, che mette il verde sulla strada "stretta" e così via.

Quando B = 0 si testa una delle posizioni X e se essa in precedenza conteneva il blank, viene messa la "A" nell'Accumulatore, mentre con X1 = "A" si fa il trasferimento del blank, quindi si ritorna al punto in cui A è caricato in tutte e quattro le X. In tal modo, queste stanno spente e mostrano quattro "A", alternativamente.

Vogliamo qui fare la storia di un piccolo dramma capitatoci in una prima stesura del programmino. Avevamo fatto il test dell'Accumulatore, in luogo di quello di una X. La cosa sembrava ovvia, ma lo schermo rimaneva ostinatamente spento! Solo dopo un pò ci siamo resi conto del motivo: la routine KBD, quando un tasto non è premuto, *non conserva* il valore della precedente battuta in A, ma mette in questo registro il valore 00. Perciò il confronto su detto dava sempre il medesimo risultato ed il lampeggio mancava.

Al posto dei due INC SP va altrettanto bene un POP, ad esempio di HL (ma anche un'altra coppia qualsiasi va bene, però attenti perché il POP altera il contenuto di tali registri). Lo SP subisce pure un incremento di due unità, inoltre si risparmia un byte. Per contro il duplice INC SP ha il pregio della maggior chiarezza.

Codifica

Dopo una così dettagliata descrizione del diagramma a blocchi, possiamo permetterci di limitarci, a proposito del programma codificato, alle cose più tecniche, altrimenti rischiamo oltrettutto di ripetere cose già dette.

Dobbiamo innanzitutto vedere da vicino come è fatto il sottoprogramma per il disegnino delle strade. Dopo le prime due istruzioni che servono alla pulitura del video, si pone in HL il valore 090AH, in modo da puntare all'inizio della riga TV omonima, poi in B si pone il valore 30H, cioè 48D, tanta è la lunghezza di una riga. Con le successive: LD (HL), '-' poi INC HL e DJNZ −3 si caricano su detta riga 48 trattini (codice ASCII 2DH): in linguaggio Assembly, si ricorda ancora, essi sono rappresentabili direttamente in chiaro tra apici. Seguono istruzioni analoghe precedute da un loading in HL dell'indirizzo iniziale della riga 0A0A, ossia 4 righe sotto (come chi legge potrà verificare o facendo dei non difficili calcoletti o, più direttamente, dall'esame della Fig. 3-4 cap. 3). Si caricano così trattini in tutte queste righe.

Gli altri due gruppi di istruzioni, servono a caricare caratteri "l" (in ASCII 6CH) sulle colonne il cui indirizzo iniziale è, rispettivamente, 081F e 0825. Il procedimento consiste nel mettere nella coppia DE il valore 40H, quant'è, come si è fatto a suo tempo notare, la distanza tra una riga e la seguente, in modo da utilizzare l'istruzione ADD HL,DE per fare tale passaggio (N.B. nell'aritmetica a 16 bit non esistono istruzioni nelle quali è in gioco un dato immediato). Il valore 0040H viene lasciato in DE in quanto serve più avanti.

Programma P. 6-1

Indir.	Contenuto	Label	Cod. Assembly	Commenti
		CRT	EQU 13 BH	indirizzi
		KBD	EQU 69H	della routine
		KDEL	EQU 35H	del NASBUG
		RIGAUNO	EQU 090AH	ind. 1ª riga trattini
		RIGADUE	EQU 0A0AH	ind. 2ª " "
		COLUNO	EQU 081FH	ind. 1ª colonna di '1'

Indir.	Contenuto	Label	Cod. Assembly	Commenti
		COLDUE	EQU 0825H	ind. 2ª colonna di '1' (°)
		X 1	EQU 0962H	
		X 3	EQU 09A1H	
			ORG C53H	
0C53	14	T1	DEFB 20	T1 =n° di volte RITFIX
0C54	23	T2	DEFB 35	idem T2 (verde su str. oriz.)
C55	3E 1E	SEMAFORO	LD A, 1EH	solite istruz. per la
C57	CD 3B 01		CALL CRT	pulizia dello schermo
C5A	21 0A 09		LD HL, RIGAUNO	
C5D	06 30		LD B, 30H	B = larhezza riga
C5F	36 2D	TRATT1	LD (HL), '–'	'–' = 2DH in (HL)
C61	23	INC HL		avanza una posizione
C62	10 FB		DJNZ TRATT1	carica '–' finché B = 0
C64	21 0A 0A		LD HL,RIGADUE	idem c.s. per altra
C67	06 30		LD B, 30H	riga trattini
C69	36 2D	TRATT2	LD HL, '–'	
C6B	23	INC HL		
C6C	10 FB		DJNZ TRATT2	
C6E	11 40 00		LD DE, 40H	distanza tra 2 linee in DE
C71	21 1F 08		LD HL, COLUNO	
C74	06 10		LD B, 10H	10H = 16 = n° righe
C76	36 6C	ELLE1	LD (HL), 6CH	'1' = 6CH in (HL)
C78	19		ADD HL,DE	passa a riga sotto
C79	10 FB		DJNZ ELLE1	carica '1' finché B = 0
C7B	21 25 08		LD HL, COLDUE	si ripete la solfa
C7E	06 10		LD B,10H	per la colonna due
C80	36 6C	ELLE2	LD (HL), 6CH	riempita di '1'
C82	19		ADD HL,DE	
C83	10 FB		DJNZ ELLE2	
		(continua)		

(°) - *NOTA:* così, volendo, si può riassemblare il programma solo modificando queste EQU, per avere vie di diversa posizione e larghezza.

A questo punto prosegue il programma che sostanzialmente abbiamo descritto già, onde lasciamo al lettore ormai abilissimo il compito di fare il riscontro tra flow chart e codifica, anche sulla base dei commenti segnati a fianco. Un punto che merita rilievo, oltre al POP BC in luogo di INC

Programma P. 6-1

Indir.	Contenuto	Label	Cod. Assembly	Commenti
0C85	3E 56	DACAPO	LD A, 'V'	'V' sulla
C87	CD 06 0D		CALL CAR1	via verticale
C8A	3E 52		LD A, 'R'	'R' su
C8C	CD 0E 0D		CALL CAR2	quella orizzontale
C8F	3A 53 0C		LD A, (T1)	byte all'indir. T1
C92	47		LD B,A	nel registro B
C93	CD C6 0C	ABC	CALL RITPOL	
C96	30 04		JR NC, CAMB1	se Cy=0 'A' in X1, X2 ecc.
C98	CB 79		BIT 7,C	esamina SW
C9A	28 F7		JR Z, ABC	SW=0? ancora RITPOL
C9C	3E 41	CAMB1	LD, A 'A'	'A' sulla
C9E	CD 06 0D		CALL, CAR1	strada verticale
CA1	CD 16 0D		CALL RITFIX	tienilo per 1,4 sec.
CA4	3E 52		LD A, 'R'	poi 'R' sulla
CA6	CD 06 0D		CALL CAR1	via verticale
CA9	3E 56		LD A, 'V'	e 'V' su quella
CAB	CD 0E 0D		CALL CAR2	orizzontale
CAE	3A 54 0C		LD A, (T2)	valore all'indir. T2
CB1	47		LD B,A	nel reg. B
CB2	CD 06 0C	RTPL	CALL RITPOL	
CB5	30 04		JR NC, CAMB2	se Cy=0 'A' in X3, X4 ecc.
CB7	CB 79		BIT 7,C	com'è lo SW
CB9	20 F7		JR NZ,RTPL	è "on"? ancora RITPOL
CBB	3E 41	CAMB2	LD A, 'A'	arancio su
CBD	CD 0E 0D		CALL CAR2	strada "stretta"
CC0	CD 16 0D		CALL RITFIX	per i soliti 1,4 msec
CC3	C3 85 0C		JP DACAPO	poi la giostra ricomincia
		; FINE DEL MAIN-INIZIANO SUBROUTINE		
CC6	A7	RITPOL	AND A	reset Cy
CC7	CD 16 0D	DABORD	CALL RITFIX	
CCA	CD 69 00		CALL KBD	
CCD	30 0D		JR NC,BMENUNO	se non è prem. tasto:
CCF	FE 4C		CP 'L'	B-1. Premuto 'L'?
CD1	CA DF 0C		JP Z,LAMP	se SI, va a LAMP
CD4	0E 00		LD C,0	prima di procedere, SW=off
CD6	FE 56		CP 'V'	è 'V'?
CD8	C8		RET Z	se SI, rientra (con SW=0)
CD9	CB F9		SET 7,C	altrimenti SW in on
CDB	C9		RET	e rientra
CDC	10 E9	BMENUNO	DJNZ DABORD	B-1; se =0 ancora RITFIX
CDE	C9		RET	quando B=0 rientra

(continua)

Programma P. 6-1

Indir.	Contenuto	Label	Cod. Assembly	Commenti
0CDF	C1	LAMP	POP BC	per "sgonfiare" lo stack
CE0	3E 20	SU1	LD A, ' '	blank in Acc. (°)
CE2	CD 06 0D	SU2	CALL CAR1	Acc. in X1, X2 e
CE5	CD 0E 0D		CALL CAR2	in X3, X4
CE8	06 50		LD B,80	
CEA	10 0B	DECRB	DJNZ GIU	B-1 e con B=0 va a CALL KDEL
CEC	3A 62 09		LD A, (X1)	poni in A il byte 0962
CEF	FE 20		CP ' '	cfr. con il blank (=20H)
CF1	20 ED		JR NZ,SU1	se non è, va a mett. in A blank
CF3	3E 41		LD A, 'A'	se lo è, metti in Acc.
CF5	18 EB		JR SU2	'A' e vai dove lo si mette nelle quattro X
CF7	CD 35 00	GIU	CALL KDEL	
CFA	CD 69 00		CALL KBD	
CFD	30 EB		JR NC,DECRB	se Cy=0 torna a decrem. B
CFF	FE 4C		CP 'L'	è il caratt. 'L'?
D01	28 E7		JR Z,DECRB	se SI, ignoralo
D03	C3 85 0C		JP DACAPO	se NO, daccapo!
D06	21 62 09	CAR1	LD HL,X1	punta a X1 con HL
D09	77		LD (HL), A	ponivi l'Acc.
D0A	19		ADD HL,DE	passa a X2, saltando
D0B	19		ADD HL,DE	una poi un'altra riga
D0C	77		LD (HL), A	poni anche l' Accumulat.
D0D	C9		RET	fine CAR1
D0E	21 A1 09	CAR2	LD HL,X3	punta a X3
D11	77		LD (HL), A	A in (HL)
D12	23		INC HL	passa a due
D13	23		INC HL	posizioni accanto (X4) ecc.
D14	77		LD (HL),A	
D15	C9		RET	fine CAR2
D16	AF	RITFIX	XOR A	A=0
D17	08	EXAF	EX AF,AF'	Acc. in A'
D18	CD 35 00		CALL KDEL	rit.do 5 msec. c.ca
D1B	08		EX AF,AF'	ricupera vecchio A
D1C	3D		DEC A	togli uno
D1D	C8		RET Z	esci se A = 0
D1E	18 F7		JR EXAF	se no va a ri-salvare A ecc.
			END	fine fatica.

(°) - *NOTA:* il blank (esadec. 20) nel sorgente si rappresenta anche con due apici che racchiudono ... il vuoto (così avviene sempre nella *stampa* del listing).

SP INC SP, peraltro già anticipato, è rappresentato dalle ripetute chiamate delle routine CAR1 e CAR2, che provvedono a caricare "V" o "R" o "A" su in X1 e X2 e, rispettivamente, in X3 e X4. Questo modo di procedere, oltre che a risparmiare qualche byte, serve a dare un'impronta più generalizzata al programma: modificando CAR1 e CAR2 si può pensare ad un diverso dispositivo di emissione ecc. Descriveremo più avanti tali routine e la RITFIX, che presenta una particolarità.
(seguito codifica Programma 6-1)

Prima di proseguire col resto, facciamo due osservazioni: 1) quando in un programma non corto si hanno molte label occorre far attenzione a che *tutte siano diverse* e non lasciarsi andare alla tentazione di utilizzare stesse label in parti molto simili (come qui si verifica: v. label TRATT1 e TRATT2 o CAMB1 e CAMB2, cui un distrattone avrebbe dato un medesimo nome; 2) ce n'eravamo dimenticati: anche se le righe saltate tra le linee coi trattini sono in numero minore delle corrispondenti colonne con le "I", la via verticale appare più stretta, in quanto tra un carattere e l'altro di una riga c'è, fisicamente, uno spazio inferiore che tra una riga e l'altra.

Osservazioni sulle ultime routine

CAR1 punta inizialmente alla posizione X1 e pone ivi il carattere (precedentemente caricatovi, nel MAIN) sito nell'Accumulatore. Per passare alla posizione X2, posta due linee sotto, in verticale, occorre aggiungere due volte 40H (= 64D, cioè 48 posizioni visibili più 16 non visibili). Ciò si fa con un duplice ADD HL,DE ed ecco visto perchè in DE è stato lasciato il valore 40H, già servito nella parte iniziale (disegnino strade).

Quanto alla routine CAR2, che serve a caricare in X3 ed X4, le operazioni procedono in modo perfettamente analogo, solo che qui, essendo queste posizioni sull'orizzontale, al posto dei due ADD HL,DE c'è ora un doppio INC HL (per lasciare uno spazio tra le due).

Infine la RITFIX comprende la particolarità che, prima di chiamare KDEL si salva in A' l'Accumulatore e lo si recupera in uscita da KDEL, prima di decrementare A. Questo perchè in uscita da KDEL è A = 0. È una finezza che potrebbe andar bene in casi in cui tutti gli altri registri sono impegnati. Qui andava meglio il solito uso di B inizialmente posto a zero, seguito da: CALL KDEL poi DJNZ −3 e RET (lunghezza 8 byte anzichè 10). In entrambi i casi, si ha un ritardo pari a 256 volte quello di KDEL.

Altre considerazioni

Nel corso della discussione abbiamo incontrato il problema del tasto premuto troppo a lungo e degli inconvenienti che ne derivavano (monopolio di verde di una via). Si può dare di esse una soluzione hardware ed una software. La prima consiste in un dispositivo elettronico ("*one shot*" o simi-

li) in grado di rispondere al solo fronte di salita del segnale (°), in modo che l'impulso risultante abbia sempre la medesima durata (v. Fig. 6-2 a)

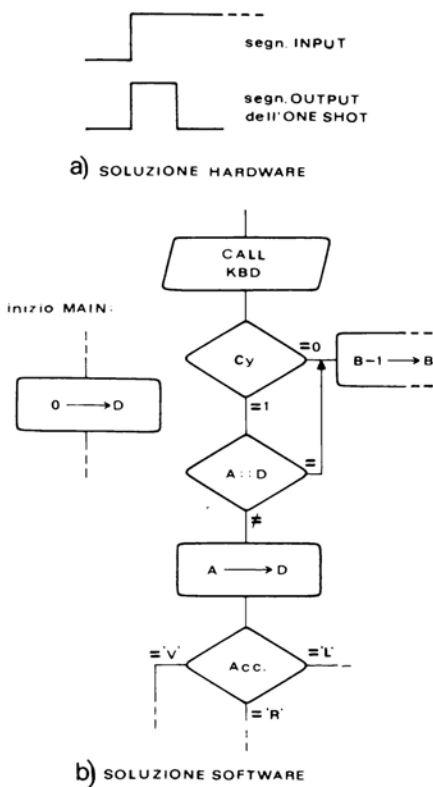



Figura 6-2

La soluzione software è illustrata nella stessa figura, in b. Si tratta di inserire (routine RITPOL, v. flow di Fig. 6-1) a valla della condizione Cy = 1 il confronto di A con D, dove D funge da deposito per la *precedente* battuta. Se A è *diverso,* ciò vuol dire che si tratta di un comando *nuovo*, quindi esso è accettato dopo essere depositato in D, per confronti successivi del genere

(°) - Qui è implicito il riferimento al caso di linee di comando singole, con pulsanti, in arrivo sui bit di un porto di I/O, come del resto è più logico in una applicazione reale.

che abbiamo visto. Se invece A e D concordano, si ignora questo comando (in quanto *già accettato* in precedenza) e si va al normale decremento di B, *come se Cy fosse 0.* Affinchè però inizialmente non si abbia l'inconveniente che il primo comando venga ignorato, occorrerà che, nella parte introduttiva del MAIN, venga posto in D un valore qualsiasi (es. 0), purchè diverso da "V", "R" o "L".

Subroutine. **In questo programma si vedono un po' di subroutine. È il** momento buono per parlare in generale della loro convenienza. Detto subito che ne guadagna in ogni caso la compattezza (e chiarezza) del MAIN, vediamo entro quali limiti una CALL riduce il numero di istruzioni. Occorre che la chiamata sia fatta almeno *due* volte altrimenti si aggiungono 4 byte: i 3 della CALL più quello della RET. Per due chiamate, dato che le istruzioni aggiunte occupano $3 \times 2 + 1 = 7$ byte, occorre che il numero minimo di byte della subroutine (RET esclusa) sia 7 (vantaggio nullo, mentre con 8,9 ecc. si guadagna uno, due ecc. byte). Ragionando analogamente, si trova: con 3 chiamate, il numero minimo è 5; con 4 è 4 ecc.

Bisogna poi vedere caso per caso. Può infatti capitare ad esempio che la soluzione senza chiamate di routine comporti, rispetto a quella con l'impiego di esse, necessità di salti vari che la rendono, oltrechè più involuta, anche più costosa in termini di byte, di quanto non si sarebbe detto sulla base di una meccanica applicazione delle regolette viste.

Infine, a proposito delle subroutine a uscite differenziate, noi abbiamo escogitato il trucco del JP senza RET (più il POP BC o simili per aggiustare lo stack pointer). In altri casi, specie se tali rientri differenti sono più di uno si possono utilizzare o deviatori (a più vie, magari) o, più razionalmente, l'istruzione JP (HL) - salto indiretto - posta al rientro dalla routine (N.B. non *dentro* essa, a meno di non voler usare il nostro trucco osceno ...). Dentro la routine, nei punti più adatti, si avrà cura di porre in HL il valore opportuno. Abbiamo voluto parlare, per completezza, di questa raffinata tecnica, anche se negli esempietti nostri non abbiamo occasione di applicarla.

Programma P. 6-2

OROLOGIO-CRONOMETRO

Si tratta di un programma niente affatto originale, è solo un tributo che paghiamo al desiderio di completezza. Comunque, anche se lo svolgimento è piuttosto elementare e lineare, l'adattamento alle specifiche del NASCOM e qualche altro particolare lo rendono non del tutto banale.

Ne daremo soprattutto una descrizione tramite flow chart, lasciando al lettore desideroso di vederlo funzionare il gravame di gran parte della minutazione del sorgente. Riferiamoci pertanto alla Fig. 6-3.

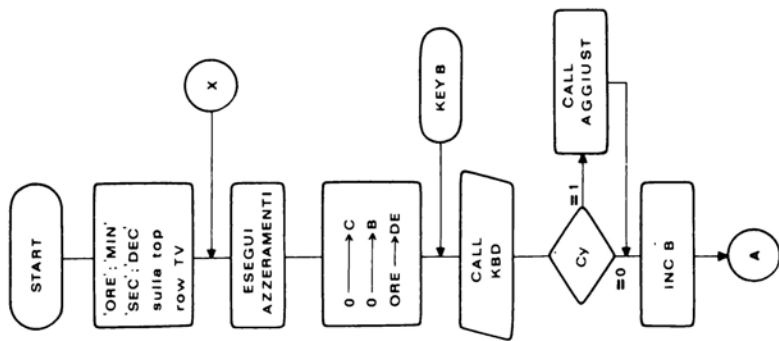
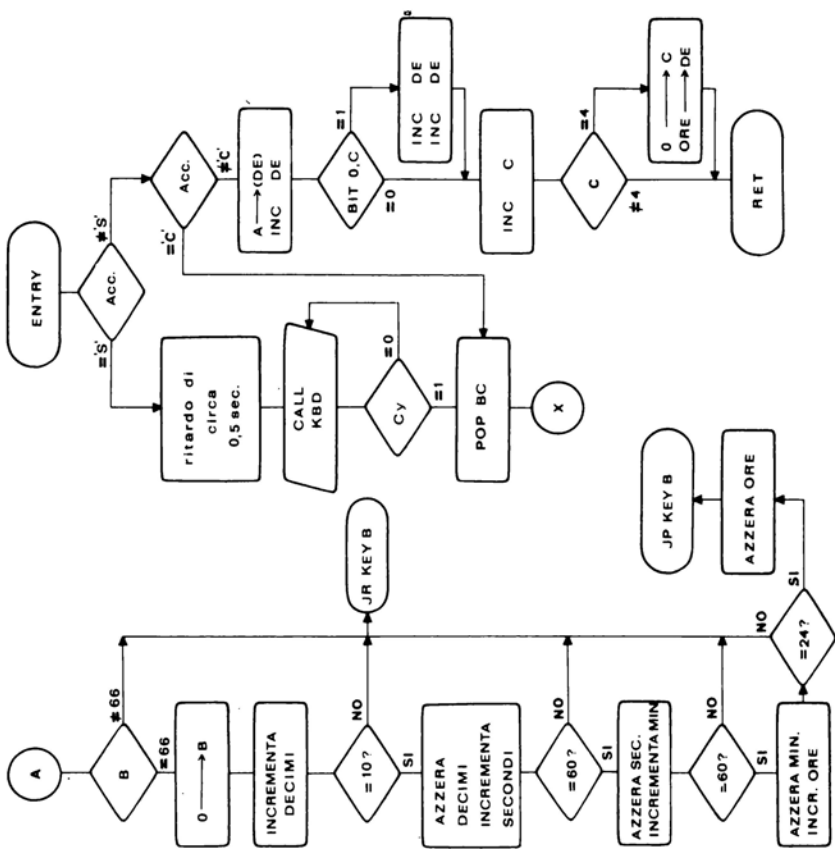



Figura 6-3 Orologio-cronometro.

Vogliamo anzitutto far apparire le diciture "ORE"; "MIN"; "SEC" e "DEC", come intestazioni di ovvio significato, sulla top row. Si possono scegliere gli indirizzi 0BDA per la "O" di "ORE" e, per l'inizio delle altre tre diciture, rispettivamente 0BCE, 0BD2 e 0BD6, in modo che, cioè, uno spazio vuoto le separi. Sotto ogni dicitura andranno poi visualizzate, appunto, le ore, minuti ecc. Scegliamo la riga che inizia alla posizione 088A: le corrispondenti cifre iniziali vanno rispettivamente agli indirizzi: 0C4A (ore); 0C4E (minuti); 0C52 (secondi) e 0C56 (decimi di secondo). Alle ore 10 45′ e 28″ si vedrà insomma:

ORE	MIN	SEC	DEC
10	45	28	vv

Ove vv sta per "velocemente variabile" e quindi distinguibile solo con lo stop del cronometro. Dimenticavamo appunto di dire che vogliamo poter utilizzare il marchingegno anche come contasecondi.

Vediamo il resto del MAIN. Il blocco segnato con "ESEGUI AZZERAMENTI" è un gruppo di istruzioni che fa apparire tutti zeri sulle dette posizioni.

Quindi si azzeranno i registri C e B e con DE si punta all'indirizzo denominato con la label ORE, cioè 0C4A. Segue la chiamata della nostra cara KBD; si interroga il Carry e, se acceso, si chiama la routine AGGIUST.

A dire il vero, non v'era bisogno di una CALL, in quanto tale routine è messa in gioco una sola volta (per cui le istruzioni di chiamata più la RET sono, per così dire, di troppo). Questa impostazione ha però il vantaggio di una maggior stringatezza e, quindi, chiarezza del MAIN. L'ideale è una MACRO, se l'Assemblatore la consente.

Questa routine vedremo, serve a "regolare" il nostro orologio oppure, con la battuta del tasto "C" a farlo funzionare da cronometro.

Con Cy = 0, si prosegue incrementando B e, finchè esso non è arrivato a 20, si torna a chiamare la KBD. Lo scopo è molto chiaro: poichè la KBD, *supponiamo*, dura 1,5 msec, dopo 66 giri sono trascorsi: 100 msec. = 0,1 sec.

Qui si presenta un problema (che assilla tutti gli hobbisti e riempie le rubriche di posta sulle rivistine): poiché si hanno *circa 1,5 msec.,* come fare a rendere meno impreciso l'orologio? La risposta è: se KBD dura meno di 1,5 msec. provare ad aumentare 66 e 67 e/o aggiungere istruzioni oziose (es. NOP) ecc. Ma non vi consigliamo questo noioso e, tutto sommato, inutile esperimento: stavolta siamo in presenza di un gioco che ha l'apparenza di una applicazione, ma, almeno che si sappia, non è di impiego pratico (basti pensare ai costi).

A questo punto si azzera B e si provvede all'*incremento dei decimi* (di 1).

Se essi arrivano a 10, li si azzera e si incrementano i secondi, altrimenti (connettore con la label KEYB) si torna alla routine KBD. (°).

Con meccanismi analoghi, e di immediata comprensione sul flow, si opera con i minuti secondi, primi ecc.

Nei vari blocchi ora descritti, abbiamo messo in forma generica gli azzeramenti e gli incrementi. Questo perché si presenta il problema del codice ASCII con cui le cifre - una per posizione! - sono da mettere sullo schermo, ossia 30H, 31H, 32H ecc., per rappresentare le cifre 0, 1, 2 e così via. Occorre aggiustarsi in una qualche maniera. Noi abbiamo proceduto in un modo piuttosto pedissequo e semplice, che certamente è perfettibile (ma tant'è: questo problemino non ci piaceva tronpo, poi la nostra innata pigrizia ha fatto il resto).

In testa al MAIN si metteranno, ad es., gli pseudo op-code:

ORE EQU C4AH; MIN EQU C4EH; SEC EQU C52H e DEC EQU C56H

per indicare simbolicamente il primo byte delle ore, minuti ecc. sullo schermo (in tal modo, ripetiamo, un domani si decidesse di cambiare tali indirizzi, almeno il sorgente sarà da modificae solo in queste parti).

Per i blocchi di "azzeramento decimi e incremento secondi" come pure: "azzeramento secondi e incremento minuti", si è usato lo stesso procedimento, basato sull'osservazione che, in entrambi i casi, per sapere che si è arrivati a 60 (per l'esattezza: 36H e 30H in codice ASCII nelle due posizioni dello schermo) *è sufficiente constatare che la prima cifra è 36H.*

In entrambi i casi si è usata una subroutine INCR fatta nel modo che segue:

```
INCR    INC H
        LD A, 40H       termine di cfr. in Acc.
        CP H            H = 40 H?
        RET NZ          se no, rientra
        LD H, 30        altrimenti fai H = 'O'
        INC L           e aumenta di 1 L
        RET
```

Come stiamo per vedere tra pochissimo, la INCR è preceduta da una istruzione tipo: LD HL, (MIN) (o LD HL, (SEC)), che carica nella coppia HL i due byte, ad es. di indirizzo 0C4E e 0C4F. Per la solita regola, il primo finisce in L, il secondo in H. Allora, con le prime tre istruzioni, si fa il confronto di H, cioé della *cifra inferiore*, con 40H. Se si ha uguaglianza, vuol di-

---

(°) - Tra l'altro, con queste istruzioni di azzeramenti, incrementi ecc. (che nel nostro caso non sono neanche poche) l'analisi dei tempi diventa un vero rompicapo, tanto da rendere forse poco pratico anche da questo punto di vista il progetto di un orologio "preciso" a microprocessore ... Comunque non pretendiamo che la nostra opinione faccia testo.

re che con l'incremento si è passati da 39H a 40H, appunto. Se ciò non è avvenuto, si esce (RET NZ), altrimenti si pone in H lo zero ASCII, cioé 30H, si incrementa L, cioé la *cifra superiore* (in senso logico, si capisce, "date le circostanze" ...), poi si rientra.

Vediamo adesso la parte "azzera secondi e incrementa minuti", per l'altra analoga dei decimi e secondi, le cose vanno in modo perfettamente simile.

Le label che abbiamo messo non sono necessarie, hanno qui solo un carattere esplicativo, per fissare punti del programma.

```
AZZSEC   LD       HL, 3030H
         LD       (SEC), HL
INCMIN   LD       HL, (MIN)
         CALL     INCR
         LD       (MIN), HL      nuovo valore in (MIN)
         LD       A, 36H         A = '6' (ASCII)
         CP       L              è tale la cifra maggiore?
         JR       NZ, KEYB       se NO, torna a KEYB
         ...........             resto programma: azzera minuti ecc.
```

Le prime due istruzioni, chiaramente, pongono due zeri ASCII in (SEC) e nel byte a fianco (qui la storia del "low order"/"high order" non conta, dato che H = L). La parte INCMIN, inizia, come preannunciato, con il caricamento in HL della voce (MIN), segue la subroutine testé vista, al rientro dalla quale dopo il caricamento del nuovo valore in (MIN) non resta che da fare il test del registro L: se esso contiene l'ASCII 'O', ossia 36H, si va avanti (con "AZZERA MINUTI INCREMENTA ORE", v. flow), altrimenti, sulla condizione NZ, si risalta alla solita KBD.

Incremento ore

Si procede in modo analogo, con la differenza che, arrivando a 24 si deve fare la correzione a zero. Ma lasciamo parlare il sorgente:

```
INCORE   LD HL, (ORE)
         CALL INCR         aggiungi uno alle ore
         LD A, 32H
         CP L              L = '2'?
         JR NZ, KEYB.      se non lo è, va a KEYB
         LD A, 34H         prova con '4' (se L = '2')
         CP H              è tale H?
         JR NZ, KEYB       se NO torna a KEYB ecc...
         LD HL, 3030H
         LD (ORE), HL
         JP KEYB           (termine MAIN).
```

La subroutine INCR è ovviamente la stessa di prima, che provvede all'incremento della coppia di caratteri ASCII in H e L.

Subito dopo le prime istruzioni di significato evidente, con il detto incremento della coppia HL, i successivi confronti operano in modo che si salti alla solita KEYB se anche un solo tra H e L è diverso, rispettivamente da '2' e '4' ASCII. Perciò solo se HL contiene '42' (cioé 3432H) si esegue l'azzeramento ore, quindi con il salto a KEYB *il MAIN ha termine.*

Con queste delucidazioni, il lettore è ora in grado di codificarsi tutto il MAIN (non possiamo riprodurlo perché l'editore ci lesina la carta).

Routine AGGIUST

Come si può vedere dal flow chart, si interroga l'Accumulatore, per vedere se è stato battuto il tasto 'S' (ossia "stop", nell'uso come cronometro). Supponiamo di no e che parimenti non sia 'C' il carattere ("C" indica funzionamento da cronometro). Ci troviamo quindi di fronte al caso in cui si vuole regolare il nostro orologio. La cosa procede così:

il valore dell'Accumulatore (°) è posto nella posizione puntata da DE.

Questa, si ricordi, è stata inizializzata, in principio di main, al valore ORE (= 0C4AH). Con la INC DE si passa al byte accanto, poi viene interrogato il bit 0 del registro C. Quest'ultimo, sempre in principio di main è stato posto a 0, così la prima volta, anche il bit 0 di esso è nullo. Poiché segue la istruzione INC C, al secondo CALL AGGIUST però, si ha bit 0 pari a 1. Ed ecco spiegato il semplice ma grazioso trucchetto: dato che aggiungendo 1, il bit LSB commuta continuamente da 0 a 1 e viceversa (pari, dispari ecc.), tale bit-deviatore *opera in modo automatico*, senza cioé alcuna istruzione SET o RES. A che serve, poi, tale deviatore? È semplice: al primo, terzo ecc. passaggio (passaggi "dispari") non si esegue il duplice, ulteriore INC DE, che invece viene messo in gioco nel corso dei passaggi "pari".

Ciò serve a creare i due spazi previsti tra ore e minuti.

Le successive situazioni risultano:
1° passaggio: C = 0 DE = 0C4A poi DE = C0C4B e C = 1;
2° passaggio: al rientro da AGGIUST, il registro C diventa 2 e DE aumenta di 3: DE = 0C4E, ossia, dopo la correzione della seconda cifra delle ore, ci si porta all'inizio del byte destinato alla prima cifra dei minuti;
3° passaggio: C diventa 3 e DE è incrementato solo di 1 (ci si sposta nel byte destinato alla seconda cifra dei minuti); ecc. ecc.

(°) - Che deve essere assolutamente non diverso da una cifra decimale. Il lettore perfezionista potrebbe inserire un test a tale proposito, in modo che non si accetti un valore "illegale".

Per poter correggere, successivamente (°) ore e minuti, quattro passaggi sono quelli che ci vogliono, ecco perché, quando C = 4, si provvede ad azzerarlo ed a riposizionare il valore ORE nel puntatore DE.

Si osservi che, con il rientro da AGGIUST, ci si aggancia prima del blocco in cui si incrementa B ecc., in modo che lo "scanning" della keyboard venga sempre regolarmente fatto ogni 1,5 msec., per stare all'erta alle battute che possono arrivare, ma contemporaneamente *non si blocca il conteggio del tempo.* In altri termini, mentre si indugia tra la correzione di una cifra e l'altra, *l'orologio va avanti* (inesorabile).

Resta da vedere quello che succede quando si preme il tasto 'C': dopo il POP BC, che ha la stessa funzione dell'analoga istruzione vista nell'esempio del semaforo (routine LAMP), cioé di surrogare una RET, con il connettore X ci si collega all'esecuzione di azzeramenti. Il perché è evidente, trattandosi, come si è detto, del funzionamento come cronometro.

Questo conta a partire da zero, finché, quando è pigiato 'S' si va ad un *loop d'attesa* che serve a tenere bloccato il risultato sullo schermo, in modo che il tempo intercorso tra i due eventi "C" e "S" possa essere esaminato con calma.

Questo loop è semplicemente la solita CALL KBD seguito da JR NC, –3.

Per uscire da esso basta battere un tasto qualsiasi per riprendere, dopo POP BC, tramite connettore X, dagli azzeramenti eccetera.

Si noterà un ritardo, realizzato con il solito criterio, che precede questa CALL KBD. A che serve? Dopo riflessione, non dovrebbe essere difficile rendersi conto del motivo: se questo ritardo non ci fosse, schiacciando il tasto S un po' a lungo (basta superare 2-3 msec.!) si andrebbe al punto X, cioé all'azzeramento, sicché il risultato cronometrico scomparirebbe immediatamente.

È questo un caso in cui, qualora si fosse fatto l'errore di cui stiamo parlando, alla semplice prova del programma potrebbe essere difficile accorgersi del motivo del malfunzionamento. Anche l'esecuzione passo passo, per una serie di motivi, qui non è molto pratica. Per questo si raccomanda sempre un'analisi la più accurata e *previdente possibile, in fase di flow chart.*

Si fa notare che l'attesa di 0,5 sec. o di qualunque altro tempo si tratti, non inficia il risultato, in quanto con il comando 'S' immediatamente il conteggio si blocca.

(°) - Se questa impostazione appare un po' rigida, si può studiare una variante in cui la battuta di un carattere particolare, es. la barra spaziatrice (dà in A il valore 20H), lascia immutata la cifra, sempre facendo incremetare correttamente il puntatore DE. Si può anche usare la CRT del Monitor (col comando backspace per ritornare indietro a correggere battute errate).

Una variante migliore potrebbe essere quella illustrata nella Fig. 6-4 a. Qui, senza bisogno di alcun ritardo, subito dopo la condizione Cy = 1, si interroga di nuovo l'Accumulatore e, finché esso contiene 'S' si torna a CALL KBD, cioé si resta nel loop d'attesa quantunque sia lunga la battuta del tasto 'S'. Solamente con Cy = 1 e A *diverso* da 'S' si esce dall'attesa.

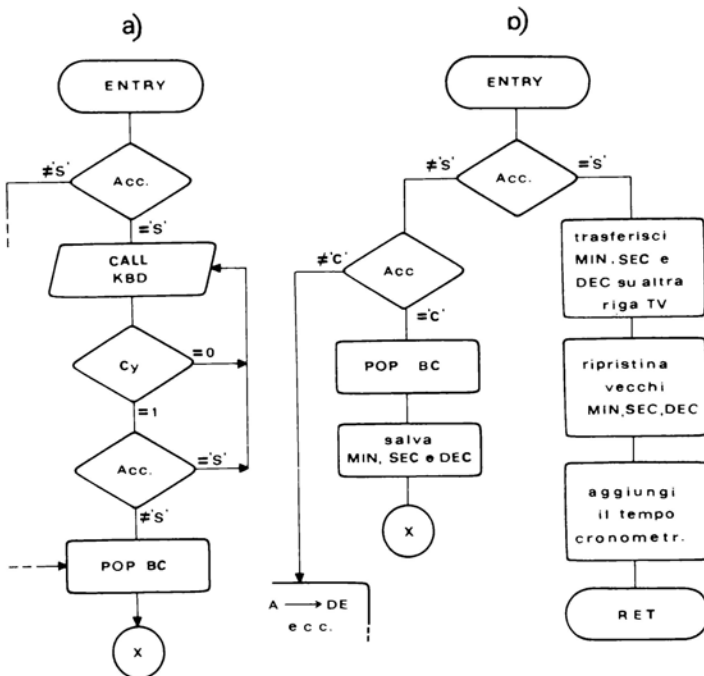



Figura 6-4 Varianti della routine AGGIUST.

Inutile dire che anche questa versione contiene ancora un'imperfezione, come molti lettori accorti avranno già capito: si tratta del fatto che, passando a funzionare come cronometro, la funzione di orologio è tolta brutalmente di mezzo e, una volta che la si vuole riprendere, c'è da rifare la regolazione. Non c'è male davvero per un aggeggio formato da tanti pezzi di elettronica ultramoderna (sembra più adatto a figurare al collo di un cannibale!).

Bisognerà, come minimo, posizionare il connettore X in modo che *non si faccia, l'azzeramento delle ore* (e magari dei minuti, quando si cronome-

trano tempi brevi). Anche così però resta la non modesta imprecisione del tempo che si perde a guardare il risultato.

La soluzione che taglia la testa al toro consiste nell'utilizzare una seconda riga per visualizzare i risultati del cronometro, *mentre l'orologio,* dopo le poche decine di microsecondi necessari per trasferire tali risultati (ore escluse) su quest'altra riga (sulla quale resteranno molto a lungo, finché non si faccia un'altra misura), *riprende subito la sua marcia* dal punto in cui era rimasto. Per la perfezione, bisognerà opportunamente salvare alla battuta di 'C' e prima dell'azzeramento, i valori dei minuti, secondi e decimi dell'orologio e, dopo la battuta dello stop 'S', far seguire il trasferimento dei dati cronometrati sull'apposita riga aggiuntiva con adatte istruzioni di ripristino dei vecchi dati dell'orologio, aggiungervi il tempo che è stato cronometrato (°) e, con una istruzione, stavolta di rientro RET, riprendere come se niente fosse accaduto (Fig. 6-4 b).

Tutte queste considerazioni possono anche apparire involute ed oziose. Abbiamo però fatto questa sorta di sceneggiata su come, per successive approssimazioni, si possa arrivare ad una soluzione migliore (non perfetta!), allo scopo di convincere come, nella programmazione vale spesso il motto dell'Accademia del Cimento: "Provando e riprovando".

A questo punto coloro che fossero desiderosi di emulare gli orologiai svizzeri, possono:

1) ridiscutere tutta la materia, escogitando eventualmente soluzioni alternative a quelle da noi lumeggiate;
2) codificare il MAIN sulla falsariga di quanto abbiamo mostrato a mo' di esempio;
3) scegliere il tipo di routine AGGIUST che si preferisce e, in base a tale scelta, codificarla, mettendola in testa o in coda al MAIN e curando che leghi bene con esso.

Quest'ultimo lavoro non presenta serie difficoltà se si segue il flow di figura 6-3 (meglio se almeno con la variante a) di Fig. 6-4), in quanto esso è già fortemente orientato al linguaggio Assembly.

(°) - Magari facendo una correzione per tener conto dei microsecondi che si perdono in tutte queste operazioni!

CONCLUSIONI

A chiusura di questo capitolo vogliamo infine fare un paio di considerazioni. Ci siamo tenuti, con questi due esempi, ancora nel quadro dei 'giochi" con il personal computer, tuttavia si tratta chiaramente di cose che hanno un significato applicativo più che evidente (sia pure con le riserve che abbiamo fatto circa il secondo esempio).

Rimane il fatto che entrambi gli esempi erano delle simulazioni.

Una volta però che se ne è capita la logica (e coloro che dispongono di sistemini di sviluppo diversi possono ugualmente adattare la codifica alle caratteristiche di ciò che hanno in casa) il passaggio al caso pratico non dovrebbe presentare difficoltà.

Più avanti daremo delle indicazioni (non troppo vaste dati i limiti del testo) sulle più semplici interfacce alternative alla solita tastiera e monitor TV che si possono connettere ad un microprocessore. Il lettore, in base a questi cenni (e/o sulla scorta di letture di manuali di hardware per microprocessori) non avrà, ne siamo sicuri, difficoltà a progettarsi l'hardware ed il software di un *reale* controller di un semaforo.

A titolo informativo, vogliamo qui accennare ad una applicazione di cui siamo a conoscenza, in cui è stato pari pari utilizzato un sistema molto simile al NASCOM, con la sola aggiunta, agli I/O port della PIO, di sensori e un paio di linee di comando. Si tratta di un sistema di sicurezza contro i furti in un ambiente che periodicamente è visitato da un guardiano notturno. Durante la sua assenza il micro fa il "polling" dei sensori, azionando eventualmente un allarme in casi alquanto sospetti, scrivendo in ogni caso di variazioni di sensori, opportuni messaggi in memoria.

All'arrivo del guardiano, questi può, battendo dei tasti, venir a conoscenza di quanto è accaduto durante la sua assenza. È evidente che qui la distinzione tra "simulazione" e "realtà" si fa un po' sottile.

CAPITOLO 7

SEMPLICI ESPERIMENTI CON LA Z80-PIO

In questo capitolo descriveremo un paio di programmi molto semplici aventi lo scopo di sperimentare sul NASCOM o su un altro sistema consimile le modalità attraverso le quali si gestiscono, con lo Z80, le operazioni di I/O e in particolare gli interrupt. Permetteremo alcuni succinti richiami di tali argomenti, invitando tuttavia il lettore che non l'avesse ancora fatto, a procurarsi il manuale Z80 e quello della PIO e a leggerli con attenzione nelle parti pertinenti.

Richiami

Il sistema Z80 tratta gli interrupt nel modo più semplice ed efficace. Esiste una priorità nella soddisfazione delle varie interruzioni:

1)	BUS REQUEST	BUSRQ
2)	INTERRUPT NON MASCHERABILE	NMI
3)	INTERRUPT MASCHERABILE	INT

ciascuna di esse proviene da una diversa linea esterna, di input, della CPU.

Il primo tipo constringe la CPU a cedere il Bus (ad esempio ad una periferica in grado di gestirsi da sola il DMA, Direct Memory Access).

Il tipo di NMI, secondo della gerarchia, non può essere in alcun modo tacitato a software, come è invece possibile con quelli mascherabili (tramite una istruzione DI). Tale richiesta genera un salto all'indirizzo 0066H. Qui dovrà essere l'inizio della relativa routine di servizio.

La sequenza delle operazioni è la seguente: la CPU memorizza sullo stack l'indirizzo dell'istruzione successiva a quella in corso (la quale è *sempre ultimata*), copia il contenuto del flip flop interno IFF1 in un secondo: IFF2, quindi azzera IFF1. Ciò disabilita gli interrupt durante la routine di servizio del NMI (tranne che il programmatore li riabiliti con una EI). Al termine di questa si ha una RETN, che provoca l'automatica ri-copiatura di IFF2 in IFF1 e parimenti il richiamo della stack del vecchio indirizzo, onde si possa ripartire dal punto di interruzione.

L'interrupt mascherabile

Può essere fatto ignorare tramite l'istruzione DI, che azzera entrambi i f.f. IFF1 e IFF2. Viceversa l'istruzione EI rimette in "on" i due consentendo di nuovo alla CPU di accettare interrupt.

Lo Z80 risponde in tre diversi modi all'interruzione mascherabile, detti Modo 0, Modo 1 e Modo 2, che si selezionano con le istruzioni IM0 , IM1 e

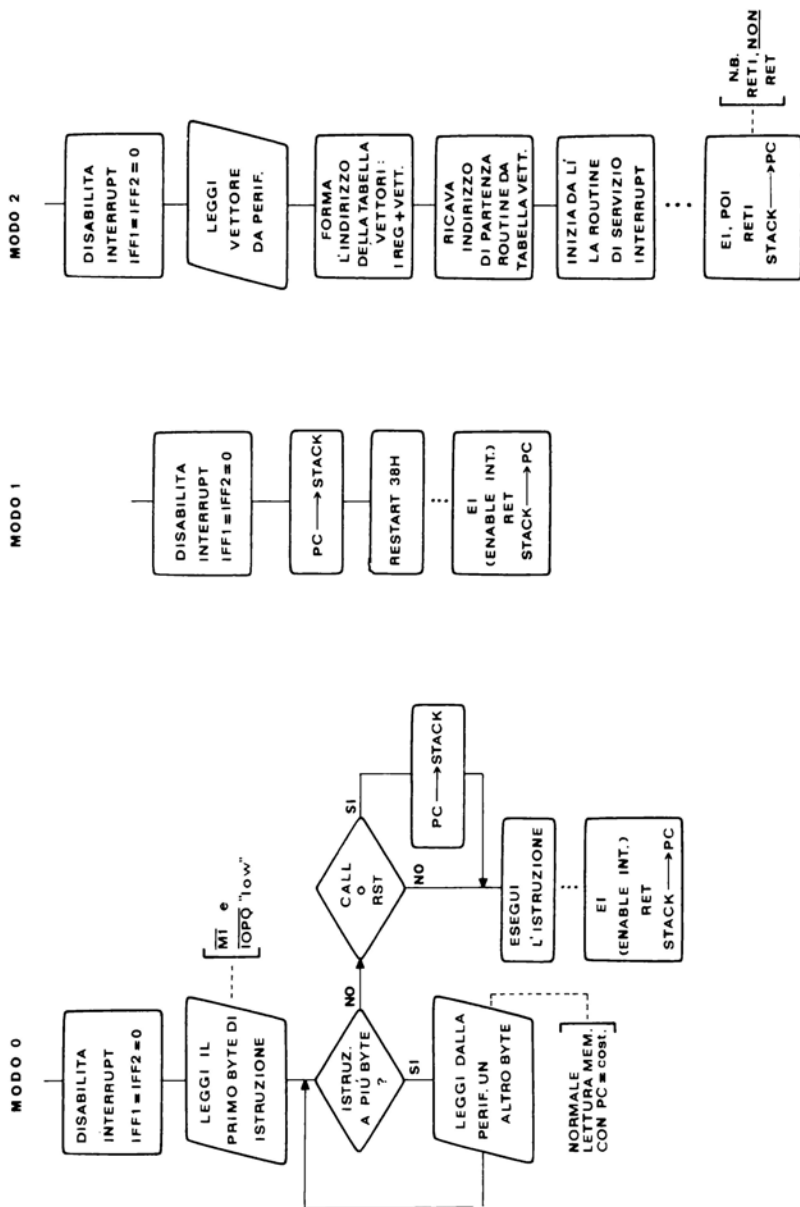



Figura 7-1 Procedure di risposta all'interrupt mascherabile.

IM2 e i cui schemi di funzionamento sono riportati nella Fig. 7-1. In tutti, IFF1 e IFF2 sono messi a zero, onde inibire il riconoscimento di altri interrupt, fino a che il programmatore, al termine quasi sempre della routine di servizio, non li riabiliti con una EI.

Modo 0: identico a quello dell'8080; con esso la periferica pone sul Data Bus l'istruzione iniziale della routine di servizio (molto spesso una RST, dato che è ad un sol byte);

Modo 1: il più semplice di tutti; la risposta è sempre un "Restart" alla posizione 38H (dove è d'obbligo iniziare la routine di risposta);

Modo 2: è il più potente. Con un solo byte di informazione, fornito dalla periferica su scelta del programmatore, si può eseguire il cosiddetto *"interrupt* vettorizzato", che consiste in un salto indiretto ad una qualsiasi posizione di memoria (in cui, al solito, inizia la relativa routine di servizio interrupt), tramite un salto indiretto. Vediamo in dettaglio come opera questo interrupt vettorizzato (Fig. 7-2).

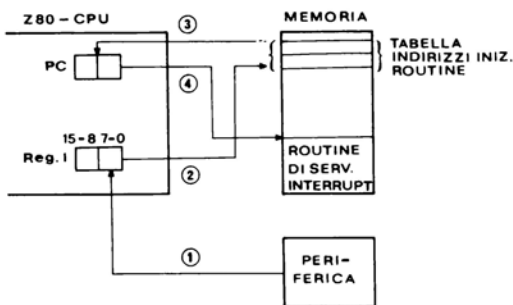



Figura 7-2 Fasi della risposta all'interrupt "vettorizzato".

In un sistema a microprocessore vi sono generalmente più periferiche, alcune delle quali sono origine di segnali di interruzione.

Il collegamento tra queste periferiche è realizzato tramite la cosiddetta "daisy chain" (v. Z80-PIO Manual) in modo che l'unità posta all'inizio di questa "ghirlanda di margherite" (traduzione letterale di daisy chain) ha la priorità su tutte le altre e che, ogni volta che un'unità di priorità maggiore richiede di essere servita tramite un interrupt, l'eventuale routine di gestione di un interrupt in corso, relativo ad unità di priorità di minor livello, viene sospesa (e ripresa al termine dell'altra) ecc.

Questo meccanismo è di grande semplificazione dei problemi di gestione di interrupt a provenienza plurima, tuttavia resta il fatto che la linea di interrupt INT della CPU è unica, onde tutte le linee di richiesta di interruzioni sono collegate, in parallelo, a questa. Come può allora la CPU operare il riconoscimento di quale tra le tante periferiche ha invitato la richiesta? A tal fine, ciascun dispositivo di interfaccia standard della serie Z80, come la PIO, la SIO, la CTC, possiede un registro interno il cui contenuto può essere definito nel programma (solitamente all'inizio).

Nel funzionamento in Modo 2, la CPU rispondendo all'interrupt signal, invia alla periferica un segnale di accettazione della richiesta (formato dall'attivazione simultanea delle linee di uscita IORQ e M1); questa, per farsi riconoscere, manda a sua volta alla CPU il contenuto di quel suo registro interno, detto "*vector register*", sul Data Bus.

Questi 8 bit rappresentano i meno significativi di un indirizzo di memoria, i cui 8 bit più "pesanti" sono contenuti del registro I della unità centrale (°). Per esprimersi con le parole del già citato testo di Rubino e Zaccaria (v. Bibliografia): *"L'insieme di questi 16 bit viene interpretato quindi come l'indirizzo di una cella di memoria che contiene a sua volta (insieme alla cella successiva) l'indirizzo di inizio del sottoptogramma che soddisfa la richiesta della periferica. Questo tipo di indirizzamento indiretto sembra macchinoso, ma permette di tenere in memoria una tabella con gli indirizzi di inizio di tutti i sottoprogrammi di servizio per gli interrupt relativi alle varie periferiche"*.

Ciò consente anche la modifica dinamica della routine di servizio di una data periferica, semplicemente cambiando un indirizzo della tabella.

L'unità Z80-PIO (Parallel Input Output)

Si tratta del dispositivo di interfaccia parallelo - cioè serve a trasmettere e ricevere dati di 8 bit simultanei - *programmabile* ed è senz'altro il più tipico ed importante. Esso consente agevoli collegamenti alle periferiche più disparate: tastiere, unità a banda perforate, stampatrici, programmatori di EPROM o qualunque combinazione di linee indipendenti fino ad un massimo di sedici.

Non occorre alcun altro circuito logico esterno.

Senza riportare il diagramma a blocchi della struttura interna di una PIO, ricordiamo brevemente che essa consta di due porti bidirezionali a 8 bit, denominati *porto A* e *porto B*, entrambi forniti di linee di "*handshake*": ready e strobe, operanti in uno dei seguenti modi, tutti programmabili:

(°) - Come si vedrà dagli esempi, è il programmatore che fissa inizialmente il valore di questo registro I.

- ingresso; - uscita; - bidirezionale (vale solo per il porto A) e - controllo dei bit (è quello che applicheremo di più negli esempi).

All'interno della PIO, oltre ai due porti A e B, collegati verso l'esterno, vi sono le linee di collegamento al Data Bus della CPU, circuiti logici interni, relativi anche al controllo degli interrupt, con il "traffico" interno di dati che passa attraverso un Bus interno.

Struttura interna di un porto (A o B)

Comprende i seguenti registri, nonché la logica di controllo dei segnali handshake (°):

- *registro di ingresso dati* (8 bit) tramite il quale si ha il trasferimento dati dalla periferica alla CPU;
- *registro di uscita dati* (pure 8 bit) per operare il trasferimento opposto, dalla CPU alla periferica;
- *registro di controllo del modo* di funzionamento di un porta A o B), a 2 bit, permette alla CPU (cioè al programma cui essa obbedisce), di definire uno dei possibili modi di operare, e cioè:

Output	o modo 0
Input	o modo 1
Bidirezionale	o modo 2
Controllo bit	o modo 3

- *registro di selezione Input/Output* (8 bit), nel quale la CPU carica un dato che definisce ogni linea del porto come ingresso o uscita;
- *mask register* (8 bit) il dato ivi immesso dalla CPU stabilisce quali bit del porto vanno tenuti in conto e quali ignorati negli interrupt relativi al "controllo mode" (o modo 3); per l'esattezza, sono ignorate le linee che corrispondono agli "1" di questo registro;
- *mask control register* (2 bit): questi due bit specificano se come stato "attivo" delle linee debba essere assunto lo 0 oppure l'1 e se l'interrupt debba essere generato quando *tutti* i "pin" non mascherati sono attivi (condizione AND) oppure se basta che uno solo (o più) di essi lo sia (condizione OR).

Gli ultimi *tre* registri sono esclusivi del modo di funzionamento 3. Riepilogando, con questo modo si genera un interrupt ogni volta che sono attivi

(°) - Lett. "stretta di mano". Sono due linee, RDY e STB, tramite le quali avviene il dialogo tra PIO e periferica (con RDY la PIO si dichiara *pronta*, alla trasmissione/ricezione, con l'altra è la periferica a preavvertire l'operazione. Anche questo discorso, avvertiamo, esula dagli scopi degli esempi preannunciati.

(nel senso specificato del registro di mask control) *i bit corrispondenti agli zeri* del mask register. Il registro di selezione I/O, sempre in questo modo 3, stabilisce quali bit sono da considerarsi di input e quali di output.

È questo il modo di funzionamento più utile nelle applicazioni correnti di controllo.

Può ad esempio avvenire che vi siano tre linee di comando verso l'impianto e che da esso prevengano due linee di rilevamento chiamiamole "normali", mentre altre tre corrispondono, supponiamo a segnali di allarme (massima temperatura, massima pressione ecc.). Nel registro di selezione I/O andranno allora specificate quali delle 8 linee sono di input e quali di output; nel mask register verranno "mascherati" (con dei valori "1" n.b.) i bit delle linee diverse dalle ultime tre; infine nel mask control register di fisserà, ad es. il valore "1" come stato attivo (°) e si preciserà la condizione OR per le linee di interrupt.

Piedini del "chip" Z80-PIO

Facciamo riferimento alla relativa figura del manuale Z80-PIO. Essi sono:

D_0-D_7 connessi al Data Bus della CPU, bidirezionale;
$\overline{CE}$ = Chip Enable, attivo "low", abilita la PIO;
Φ, clock che pilota la CPU e tutte le interfacce come la PIO ecc. del sistema, assicurandone il perfetto sincronismo;
$\overline{M1}$, di ingresso (attivo "low") connesso alla omonima linea della CPU, comunica che questa sta eseguendo il ciclo macchina M1;
$\overline{IORQ}$, ingresso, richiesta dalla CPU di operazioni di I/O;
IEI e IEO linee usate nei collegamenti in "daisy chain";
$\overline{INT}$ di uscita, usato dalla PIO per richiesta di interrupt alla CPU (connesso alla omonima linea di questa);
A_0-A_7 bus bidirezionale dei dati del porta A;
ARDY di uscita, con questo valore = 1 la PIO comunica che il porto A è pronto per l'operazione richiesta;
$\overline{ASTB}$ di input, è la linea tramite la quale la periferica connessa al porto A "informa" la PIO dell'esecuzione dell'operazione richiesta: nel control mode questo segnale è inibito internamente, mentre quello ARDY è disattivato e tenuto forzatamente basso;

(°) - Un utente potrebbe trovare personalmente convenienza a definire attivi gli stati "1" in tutte le sue applicazioni; dovrà porre allora degli *inverter* nei casi in cui ciò non accade. Rinciando a tale uniformità, gli inverter non servono.

vi sono poi le analoghe linee B_0-B_7 dei dati e BRDY, BSTB di handshake del porto B.
Infine:
PORT B/A SEL, di ingresso, con il valore 1 seleziona il porto B, con quello 0 il porto A;
CONTROL/DATA SEL (o più brevemente, C/D SEL), di input; con il valore 1, il byte presente sul data bus D_0-D_7 è interpretato come "*parola di controllo*", mentre con il valore 0 esso è considerato un normale dato relativo ad uno dei due porti A o B (a seconda del valore che in quel momento ha la linea B/A SEL).

Queste due ultime linee servono all'indirizzamento insieme alla CE, della PIO (e di ogni altra unità costruita con questa filosofia, come la SIO ecc.) della famiglia Zilog. Sono linee destinate di norma ad essere connesse all'Address Bus della CPU: la CE, tramite di solito un dispositivo selettore, sceglie una determinata PIO, mentre le linee B/A SEL e C/D SEL sono, come ad esempio avviene nel nostro NASCOM, collegati alle linee A_0 e A_1 dell'Address Bus. Dalla CPU vengono così indirizztai *quattro* "*porti*" con gli indirizzi (nel NASCOM) 4, 6, 5 e 7. Il primo corrisponde a B/A = C/D = 0 e individua il porto A; il secondo ha B/A = 0 ma C/D = 1. Per la CPU si tratta sempre di un porto di I/O, anche se ora in realtà è la "parola di controllo" del porto A ad essere in gioco. Comunque è tramite questo porto 6 che si *programma* la porta A della PIO. Analogamente, si ha il porto 5 per B e il porto 7 per la sua parola di controllo.

Programmazione della PIO

Mandando al porto di controllo delle parole *aventi un formato opportuno,* si programma il funzionamento della PIO. Tali parole di controllo sono:

- *Caricamento del vettore di interrupt*
Come già detto parlando di interrupt, nel funzionamento in Modo 2, il dispositivo che interrompe deve fornire gli 8 bit meno significativi dell'indirizzo della tabella delle routine di servizio. Questi bit costituiscono il *vettore di interrupt*, il cui formato è (°):

$$V_7\ V_6\ V_5\ V_4\ V_3\ V_2\ V_1\ 0$$

in altri termini, ogni volta che una parola che ha il bit LSB *nullo* è inviata (dalla CPU, n.b.) a un porto di controllo, da quel momento in poi il porto ("*vero e proprio*", A o B che sia), al verificarsi delle condizioni di interruzio-

(°) - Qui e di seguito un formato è rappresentato da otto simboli ognuno dei quali designa uno degli 8 bit della control word.

ne, si fa riconoscere dalla CPU inviando tale vettura. Questo vattore, come pure altri elementi relativi alla programmazione di un porto, è, almeno nella maggior parte delle applicazioni, predisposto "*una tantum*", all'inizio del MAIN program.

- *Modo di operazione*

Questa parola di controllo è caratterizzata dal valore esadecimale F nel semibyte di destra e il suo formato è:

$$M_1\ M_0\ X\ X\ 1\ 1\ 1\ 1$$

Ove i bit contrassegnati con X sono "don't care" cioè non conta il loro valore, mentre $M_1\ M_0$ valgono 00, 01, 10 e 11 per i modi, nell'ordine, "0" = Output, "1" = Input, "2" = Bidirezionale, e "3" = Controllo dei bit. (°)

Quando viene scelto il modo 3 è necessario il successivo invio al porto di controllo di una parola (che viene caricata nel registro I/O Select) con la quale viene precisato quali sono le linee scelte come ingressi e quali quelle assunte come uscite. Questa parola ha il formato:

$$I/O_7\ I/O_6\ I/O_5\ I/O_4\ I/O_3\ I/O_2\ I/O_1\ I/O_0$$

da inviare obbligatoriamente subito dopo la precedente. Questo obbligo è tipico delle parole di controllo prive di bit di codice. Il valore I/O = 1 caratterizza il bit corrispondente come linea di input, mentre I/O = 0 lo qualifica come di output (sempre relativamente al porto A o B, cui la parola di controllo si riferisce).

- *Controllo dell'interrupt*

Il codice di questa parola è l'esadecimale 7 nel semibyte destro. Il formato è schematizzabile come segue:

IE A/O H/L MF 0 1 1 1

IE significa Interrupt Enable, cioè l'interrupt eventualmente proveniente dal quel certo porto è abilitato de IE = 1, mentre con IE = 0 è disabilitato. Attenzione a non confondere questo comando (implicito) che si riferisce ad *un* porto particolare (quello cui, al solito, il porto di controllo è associato) con il comando EI (e l'opposto DI) che vuol dire abilitazione (disabilitazione) di *tutti* i possibili interrupt, da qualunque porto provengano.

Gli altri tra bit del semibyte sinistro, A/O H/L, MF sono usati solo nel modo 3 (e ignorati negli altri modi): con A/O = 1 si definisce l'operazione

---

(°) - Si noti come i numeri siano stati dati in funzione mnemonica: "0" = O. "1" = I. e "2" = due direzioni. Attenzione poi a non confondere il modo di funzionamento di un porto con l'Interrupt Mode (IM) della CPU. Il modo 3 del porto è associato all'IM 2.

logica AND, con lo stesso bit = 0, si definisce l'operatore OR, tra i bit dell'interrupt, con H/L = 1 si fissa come stato "attivo" il valore "High", mentre il valore "Low" corrisponde ad H/L = 0; infine MF = 1 denota "Mask Follow" e vuol dire che la successiva parola è una maschera, che andrà quindi caricata nel mask register. Se la maschera segue, essa ha il formato seguente:

$$MB_7\ MB_6\ MB_5\ MB_4\ MB_3\ MB_2\ MB_1\ MB_0$$

i cui bit indicano: MB = 0, bit da controllare (per generare l'interrupt);
MB = 1, bit da ignorare

Infine è anche possibile abilitare o disabilitare *in qualsiasi momento* l'interrupt di un certo porto con l'invio di una parola avente il semibyte destro uguale a 3H, con il formato:

IE X X X 0 0 1 1

È quasi inultile dire che: X = bit "don't care"; IE =1 abilita e IE= 0 disabilita

Un pochino di hardware

Dopo tutti questi noiosi ma indispensabili richiami, è tempo di compiere esperimenti molto semplici ma che servano a capire come operano le istruzioni di I/O e gli interrupt. Per questi ultimi ci limiteremo al caso più importante (e tipico dello Z80), caratterizzato dall'IM 2 e dal modo 3 di funzionamento del PIO Port (A o B).

È indispensabile cominciare a vedere l'input/output tramite porti, perché, nei progetti reali di controllo raramente si dispone di tastiera alfanumerica come abbiamo fatto finora negli esperimenti di simulazione.

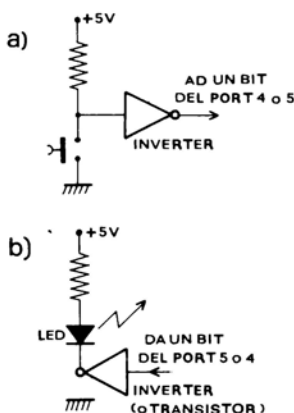



Figura 7-3 Connessione interruttori e LED ai porti di I/O.

A tale scopo, realizziamo su una piccola scheda un circuito semplicissimo (a un punto tale da non meritare quasi di essere illustrato da una apposita figura). Esso sarà costituito da otto interruttori e/o pulsanti (per es. 4 di un tipo e 4 dell'altro) che ci serviranno come linee di ingresso (manuale).

Essi possono collegare al porto 4 della PIO del NASCOM ciascuno come in Fig. 7-3 a, tramite inverter, in modo che, con il pulsante (o tasto) azionato venga trasmesso il bit "1". Sulla stessa schedina disporremo 8 diodi emittenti luce (LED), per esempio collegati al porto 5 anche qui tramite invertitori (°), in modo che il bit di uscita "alto" di tale porto faccia accendere il LED.

Ricordiamo poi di nuovo che i porti di controllo sono, per la PIO del NASCOM:

- porto 6 di controllo per il porto 4 e
- porto 7 di controllo per il porto 5.

Con l'hardware realizzato è implicito che definiremo il porto 4 come input e il porto 5 come output. Per definire e studiare il modo 3 potremo, in alternativa, porre dei LED su alcune linee, ad esempio, del porto 4.

Prima di descrivere gli esempi preannunciati, facciamo due considerazioni:

1) Naturalmente sarebbero possibili esperimenti ancora più elementari, consistenti nel collegare *direttamente,* con pulsanti o altro, limitandosi ad interporre dei dispositivi "latch" per i problemi di memoria. Tuttavia, dato anche il costo relativamente basso di dispositivi di interfaccia tipo PIO, raramente, in progetti reali, si rinuncia ad uno o più di tali dispositivi, con tutta la loro potenza e programmabilità; beninteso, anche i nostri esempi sono del tipo, banalissimo, "accendi-la-lucetta-spegni-la-lucetta", ma, almeno, serviranno a chiarire l'uso della PIO e degli interrupt.

2) La PIO "sciupa" due porti per il controllo, ma questa limitazione del numero delle periferiche interfacciabili comincia a prendere consistenza soltanto nei progetti molto complicati. Si pensi che il numero di porti (veri e propri e/o di controllo) indirizzabili dalla CPU arriva a ben 256!

A questo punto è tempo di passare all'esempio numero uno.

(°) - Per risparmiare, si possono limitare tasti e LED a sei, anziché otto o dieci. Così facendo necessitano solo due integrati per gli inverter.

Programma P. 7-1

GESTIONE DELLA PIO - MODO 3 (listato a pag. 148)

Il programma è stato strutturato in modo che precedono delle subroutine poi impiegate nel MAIN: una routine DELAY di ritardo variabile a seconda del valore caricato nella coppia BC prima della chiamata di DELAY, poi un sottoprogramma INIPIO (= Inizializzazione della PIO) che è la parte comune all'esempio denominato ITEMUNO in cui non si usa l'interrupt e all'altro, chiamato INTEMDUE, in cui l'interrupt viene utilizzato.

Sulla routine DELAY non spendiamo altre parole, essendo già stata vista altre volte.

Routine INIPIO

Con le istruzioni: LD A,0FH poi OUT (7),A si scarica sul porto di controllo 7, relativo al porto di output 5, la parola avente i bit più leggeri tutti a "1". Ciò la qualifica come termine di definizione del modo di operazione e, poiché i primi due bit di essa sono nulli, ciò indica che il porto 5 è di output.

Da un punto di vista formale, avremmo potuto definire il modo 3 anche sul porto 5, indicando per esso tutti i bit come bit di output, ma naturalmente la via che abbiamo scelto è più semplice, oltreché più logica.

Le istruzioni che seguono: LD A,FFH quindi OUT (6), A, fissano il modo 3, di controllo, sul porto 4, anche qui tramite una parola di comando individuata dalla "F" nel semibyte "low order".

Si hanno *subito dopo* istruzioni che precisano quali bit sono, nel porto 4, di input e quali di output: la parola FCH caricata sul solito "porto" 6 (°) dice che i bit 1 e 0 del porto 4 vanno intesi come *output* (vi sono collegati, come si è supposto, dei LED), mentre gli altri sono di input.

Routine INTPIO

Comprende le istruzioni specifiche dell'interrupt.

Si inizia caricando nellà coppia HL il vettore di interrupt che abbiamo chiamato TABINT. Questa label è stata associata, all'inizio del sorgente Assembly, alla voce situata all'indirizzo 0C50. È interessante rimarcare come il contenuto di tale voce, ossia, come si è detto, l'indirizzo iniziale della routine GESTIN (cioè 0CA5) sia stato designato nel sorgente solamente col

(°) L'istruzione AND FCH equivale perfettamente a LD A,FCH.

Programma P. 7-1

Indir.	Contenuto	Label	Cod. Assembly	Commenti
.		KDEL	EQU 35H	
			ORG C50H	
OC50	A5 0C	TABINT	DEFW GESTIN	vettore di interrupt
C52	CD 35 00	DELAY	CALL KDEL	routine
C55	0B		DEC BC	di
C56	AF		XOR A	ritardo
C57	B8		CP B	variabile
C58	20 F8		JR NZ,DELAY	
C5A	B9		CP C	
C5B	20 F5		JR NZ,DELAY	
C5D	C9		RET	fine DELAY
C5E	3E 0F	INIPIO	LD A,0FH	mode 0 su
C60	D3 07		OUT (7),A	porto 5 (di output)
C62	3E FF		LD A,FFH	mode 3 su
C64	D3 06		OUT (6),A	porto 4, (control m.)
C66	E6 FC		AND FCH	bit 1,0 = out. altri bit
C68	D3 06		OUT (6),A	= inp. sempre su porto 4
C6A	C9		RET	fine INIPIO
C6B	21 50 0C	INTPIO	LD HL,TABINT	caric.to del vettore
C6E	7D		LD A,L	di interrupt su
C6F	D3 06		OUT (6),A	porto 4
C71	3E 37		LD A,00110111B	carica parola di
C73	D3 06		OUT (6),A	controllo interrupt
C75	3E 0F		LD A,0FH	"monitored" bit: 7, 6, 5, 4
C77	D3 06		OUT (6),A	
C79	C9		RET	fine INTPIO
		; ESEMPIO APPLICATIVO SENZA INTERRUPT:		
C7A	CD 5E 0C	ITEMUNO	CALL INIPIO	
C7D	DB 04	LUCINE	IN A,(4)	porto 4 su
C7F	D3 05		OUT (5),A	porto 5
C81	01 F4 01		LD BC,500	ritardo
C84	CD 52 0C		CALL DELAY	di circa 2,5 sec.
C87	AF		XOR A	quindi buio
C88	D3 05		OUT (5),A	su p. 5
C8A	01 FA 00		LD BC,250	per
C8D	CD 52 0C		CALL DELAY	1/2 tempo precedente
C90	18 EB		JR LUCINE	torna a riaccendere lumi
		; ESEMPIO APPLICATIVO CON INTERRUPT:		
C92	CD 5E 0C	ITEMDUE	CALL INIPIO	
C95	CD 6B 0C		CALL INTPIO	
C98	7C		LD A,H	bit "pensanti" di TABINT
C99	ED 47		LD I,A	nel registro I
C9B	ED 5E		IM2	poni modo 2 di interr.
C9D	FB		EI	abilita interrupt
C9E	3E F3		LD A,F3H	abilita l'interrupt
CA0	D3 06		OUT (6),A	su porto 4
CA2	C3 A2 0C	STABON	JP STABON	attendi l'interrupt

Indir.	Contenuto	Label	Cod. Assembly	Commenti
		; SEGUE ROUTINE DI SERVIZIO INTERRUPT:		
CA5	DB 04	GESTIN	IN A,(4)	filtro software:
CA7	57		LD D,A	Acc. in Deposito
CA8	01 32 00		LD BC,50	attendi circa
CAB	CD 52 0C		CALL DELAY	250 msec.
CAE	DB 04		IN A,(4)	rileggi dal porto 4
CB0	BA		CP D	è ancora valore prec.?
CB1	20 0B		JR NZ,FINEINT	se NO fine interrupt
CB3	D3 05		OUT (5),A	se SI port 4 su port 5
CB5	01 D0 07		LD BC,2000	faccelo restare
CB8	CD 52 0C		CALL DELAY	per 10 secondi
CBB	AF		XOR A	poi spegni
CBC	D3 05		OUT (5),A	i LED del porto 5
CBE	FB	FINEINT	EI	
CBF	ED 4D		RETI	ritorna dall'interr. routine
			END	

nome assegnato a tale routine di servizio interrupt. È poi l'assemblatore che, in fase di traduzione, carica nella voce l'indirizzo giusto.

Per la parte di peso minore, cioè L, è posta nel porto di controllo del porto 4: LD A,L e OUT (6),A. A questo punto viene caricata la *parola di controllo dell'interrupt:* si pone il binario 00110111 nel porto 6, con il semibyte destro che, col valore 0111, qualifica come tale la parola di controllo, mentre il primo 0 disabilita (momentaneamente) l'interrupt; (°) il secondo 0 dice che tra i bit (precisati tra breve) che determinano l'interrupt si deve fare l'OR logico; il terzo bit, col suo valore 1 indica che tali bit sono attivi a livello "High"; infine il quarto bit 1 precisa che "*segue maschera*" (si ricordi che, ove tale bit fosse zero, *tutti* i bit sono interessati alla generazione dell'interruzione).

La maschera *deve* allora essere caricata nel porto 6 subito dopo: con le istruzioni LD A,0FH seguita dal solito OUT (6),A si indicano come bit da "tener d'occhio" (monitored) i primi quattro (N.B. i bit validi per l'interrupt sono contrassegnati con *zero* nella maschera).

Primo esempio (ITEMUNO)

Si parte chiamando INIPIO, *senza* far seguire le istruzioni della INTPIO.

(°) - A dire il vero, dato che qui c'è un solo porto da cui può provenire l'interrupt, può bastare la sola istruzione EI nel MAIN (vedi). Ma qui da un lato si è voluto illustrare ogni tipo di parola di controllo, dall'altro dare alla routine INTPIO un carattere il più generalizzato possibile.

Così facendo, lo Z80 non risponde all'interrupt (per aversi matematica sicurezza di ciò, anche in casi meno semplici di questo, si poteva inserire una DI, ma si ricorda che, all'accensione, la PIO automaticamente resetta i propri flip flop interni di abilitazione interrupt).

Il programmino consiste nell'infantile giochetto di copiare lo stato del porto 4 sul 5. I LED di questo si accendono conformemente allo stato, aperto/chiuso degli interruttori del porto 4, per poi spegnersi per metà del tempo e così via ciclicamente. Occorre un RS (o, teoricamente un iterrupt non mascherabile NMI) per uscire da questo loop, ma si può modificare il lampeggio variando lo stato degli interruttori del porto 4.

Un altro giochetto si può ottenere sostituendo la XOR A, all'indirizzo OC87, con l'istruzione CPL (assoluto 2F), che inverte tutti i bit dell'Accumulatore. L'effetto è che, se ad esempio nel porto 4 si hanno gli interruttori a due a due alternativamente aperti e chiusi, le lucette del porto 5 per 2,5 secondi sono: off-off-on-on-off-off-off-off (°), poi, per successivi 1,25 secondi: on-on-off-off-on-on-on-on e via lempeggiando.

Vale la pena, a questo punto, di precisare che un porto in "control mode" avente sia bit di input che di output può essere oggetto di istruzioni, non prive di senso pratico, sia tipo IN che tipo OUT. Con le prime, si pone in A lo stato dei bit di input, nonché quello dei bit di output relativi al valore *precedentemente* caricato sul porto, mentre con la OUT, come è peraltro logico ed evidente, si caricano nei bit di output i corrispondenti valori dell'Accumulatore, mentre i bit di input non vengono influenzati.

Una variante capace di mandare in sollucchero i patiti di queste scemenzuole, consiste in:

```
ITEMDUE     CALL INIPIO
            LD A,01H
NOELARB     OUT (5),A
            LD BC,250
            CALL DELAY
            RLCA                (ruota il bit circ.)
            JR NOELARB
```

la quale, come appare evidente all'analisi, fa ruotare i bit dell'Accumulatore e, quindi, i LED del porto di output. Senza necessità di riassemblare il tutto, dato che la variante richiede vari byte in meno dell'altra versione, basterà caricare i nuovi codici assoluti (col comando "M") aggiungendo dei NOP (assol. 00) come riempitivi (inglese "padding").

(°) - I bit 1 e 0 del porto 4, connessi a LED, sono posti, come tutti i bit di output, col reset della PIO all'atto dell'accensione dell'apparecchio, a uno. Mancando, ovviamente, ivi dei pulsanti o interruttori, questi bit possono essere messi a 0 solo tramite un'istruzione OUT (4),A. Ovvio che anche i due LED sono accesi, come pure avviene, finché non si faccia un diverso OUT (5),A, per i LED del porto 5

Altri giochi consimili potranno essere fatti, magari facendo intervenire anche i bit di output del porto in "control mode", altrimenti non si comprende bene perché li abbiano definiti come tali.

Secondo esempio con interrupt (ITEMDUE)

Stavolta, dopo le istruzioni comuni a ITEMUNO e ITEMDUE, date da INIPIO, viene invocata anche la INTPIO, dopo di che si completa il discorso del vettore di interrupt, caricando il byte H nel registro I.

È indispensabile a questo punto fare una precisazione importante: poiché la parola di controllo per il caricamento del porto 6 del vettore di interrupt (utilizzata nell'istruzione LD A,L all'indirizzo C6E, seguita da OUT (6),A ha come codice di riconoscimento l'ultimo bit (LSB) nullo *è indispensabile che ogni voce della tabella delle routine di gestione abbia indirizzo pari.* Per non sbagliare, è consigliabile definire all'inizio del programma una simile tabella, che, sia detto en passant, ma è ovvio, può avere più di una voce quando vi sono più unità interrompenti e non una sola come nel nostro caso.

L'istruzione IM2 fissa il modo 2 di funzionamento della risposta agli interrupt da parte della CPU, quindi la EI abilita la CPU all'accettazione di interrupt (non mascherabili) e, infine, la lunga sequela di istruzioni preliminari termina con: LD A,F3H seguita da OUT (6),A. Con questo, si è esaurita tutta la casistica delle parole di controllo. Il codice 3, cioè 0011B, la classifica come quella che, senza modificare altri bit della parola di controllo interrupt, col valore 0 o 1 del primo bit può, in qualunque momento, disabilitare/abilitare l'interrupt da quel dato porto. Nel nostro caso, si ha l'abilitazione. (°)

Finalmente siamo giunti al vero e proprio MAIN, che si presenta cortissimo e paradossale: un salto "su se stesso" che ha, come conseguenza, un blocco della CPU: infatti nel PC è continuamente rimesso l'indirizzo STABON e la CPU... se ne sta tranquilla. Andava anche bene un salto relativo di *zero* posizioni: JR 0 (in assoluto: 18 FE).

Qui viene spontanea la domanda: perché non fare semplicemente un HALT? Il motivo risiede nel fatto che qui si desidera, ad ogni ritorno dall'interrupt, che la CPU resti in "standby" allo stesso punto, mentre con HALT si passerebbe all'istruzione successiva (il che naturalmente può anche avvenire, in altri programmi). Si poteva volendo anche fare HALT seguita da un salto, negativo, alla HALT medesima.

(°) - È quasi superfluo il dire che, essendo i bit 6, 5 e 4 "don't care", al posto di F, sarebbe andato bene anche 8, 9, A...E. Analogo discorso vale per altri casi in cui si hanno bit indifferenti.

Deve poi essere rilavato che questa "stranezza" non dipende dalla voluta banalità dell'esempio didattico, ma la si può incontrare, almeno in certe sotto-parti, in programmi diciamo così "seri", specialmente se si hanno molte unità di input interrompenti, qualora si preveda l'intervento della CPU solamente, come di suol dire, "*sotto* (condizione di) *interrupt*", altrimenti essa deve restare inoperosa.

Routine GESTIN

Il lettore potrà modificarla a piacere. Qui è previsto inizialmente un "*filtraggio software*" che consiste nell'accettare interrupt solamente se la premuta del tasto ha una durata minima (qui 250 msec.). Caricando in A il dato del porto interrompente lo si deposita nel comodo D poi, dopo 250 msec. si rilegge dal porto 4 e, se il nuovo dato è diverso da quello precedente, in D, si bypassa il resto della routine, per uscirne. Immaginando che l'interruzione sia provocata da un qualche sensore di tipo relé di massima, ciò equivale ad ignorare valori che si estinguano rapidamente.

Non dovrebbe anche essere troppo difficile all'analisi vedere che viene accettato l'interrupt anche se la causa è intermittente ma di frequenza regolare, con periodo di ripetizione inferiore ai detti 250 msec. e "duty cicle" non troppo basso ecc. (cosa peraltro poco probabile).

Quando l'interruzione è accettata, viene simulata una sorta di allarme ottico copiando il porto 4 sul porto di uscita 5, mantenendovelo per 10 secondi circa. In tal modo le cause di interruzione di durata non infima vengono segnalate e, se la loro durata è comunque limitata lo stato, per così dire di "preallarme" cessa quasi subito. Se invece la presunta anomalia (sempre immaginando, ma solo per fissare le idee, che si tratti di evento indesiderato, il che non è detto in generale) è tale da perdurare, si entra ed esce in continuazione dalla GESTIN e l'effetto risultante è la permanenza delle luci accese sul porto di output.

Una variante non banale che proponiamo al lettore volonteroso consiste nell'aggiungere un contatore, diciamo ad esempio il registro E, che è inutilizzato. Esso dovrà essere inizializzato al valore, mettiamo, 10 con l'aggiunta dell'istruzione LD E,10 al principio della parte ITEMDUE, mentre sarà incrementato con l'istruzione DEC E, da porre *prima* della XOR A (indirizzo 0CBB). Questa istruzione sarà seguita da:

```
                JR NZ,NOFEAR      (non c'è ancora pericolo...)
                CALL MESALL       emetti allarme
                HALT              o simili (es. rit. al monitor)
NOFEAR          XOR A
                OUT (5),A
FINEINT         EI     ecc.
```

Dove, per esempio, MESALL è una opportuna routine di allarme che

potrebbe essere simulata con la visualizzazione sul TV di messaggi catastrofici a piacimento.

Per concludere, definiamo le modalità operative per questo piccolo, ma speriamo istruttivo esperimento. Dopo il solito caricamento del programma, tramite tastiera o cassetta, si dà l'"execute" con E seguito dall'indirizzo di ITEMUNO. Quando si sarà stanchi di queste lucette, si dovrà (forzatamente) agire sul tasto RS, quindi fare "E", seguito dall'indirizzo ITEMDUE, avendo cura che gli interruttori relativi ai bit 7, 6, 5 e 4 del porto 4 siano aperti, altrimenti l'interrupt scatta subito, quindi agire su uno qualsiasi di essi o su più di uno e verificare ciò che avviene, conformemente a quanto previsto dal programma.

Programma P. 7-2

ALLARME CON INTERRUPT E CRITERIO DI MAGGIORANZA

In questo esempio, che ci sembra abbastanza bellino, immaginiamo il porto 4 come input e il 5 come output. Il primo lavora in control mode, con i bit 7, 6 e 5 "monitored" per un ipotetico allarme. Si immagina che le tre linee siano collegate a tre diversi sensori però del medesimo fenomeno. Si vuole allora lanciare l'allarme che prevede anche l'arresto dell'impianto, solo quando *almeno due* tra i detti sensori sono attivati.

È questo il cosiddetto "*criterio di maggioranza*" che si applica in casi del genere, per evitare arresti troppo frequenti.

Prevediamo poi che, una volta che l'interrupt è stato accettato e dopo che la parte iniziale della relativa routine di servizio è stata eseguita (con lancio di messaggi, sirene, azionamento relé ecc. ecc.), si vuole avere la possibilità, attraverso un altro interrupt, tramite stavolta la sola linea 4 del porto 4, di ripristinare le condizioni di inizializzazione processo (per es. dopo riparazioni aut similia).

Avvertiamo, diciamo così per correttezza, che questa funzione poteva anche essere attuata con la solita tecnica del "polling", tipo:

```
POLLO       IN A,(4)
            BIT 4,A
            JR Z,POLLO
            CALL RIPRIST ecc.
```

Queste istruzioni servono a leggere e rileggere dal porto 4, finché il BIT 4 non è = 1, nel qual caso si chiama, per esempio, una routine il cui nome è un programma.

Cionondimeno non è del tutto raro il caso in cui un'azione del genere possa essere richiesta con la massima tempestività, quale solo con l'interrupt si può ottenere, senza voler considerare che, in campo didattico, le ipotesi vanno tutte bene, purché il discorso risulti istruttivo (e, aggiungiamo

sempre, non troppo pedante e barboso).

Dato il carattere un po' generale dell'esempio, non ne daremo che la versione in Assembly, per giunta incompleta delle parti relative ai vari possibili casi di allarmi, azionamenti ecc. Il solito volonteroso sarà certo in grado di completare e codificare a suo piacere.

Come si può vedere in principio, stavolta la tabella dei vettori di interrupt è...un po' meno misera del caso precedente, in quanto comprende due parole, la prima individuata dalla label VETTOR, l'altra, priva di label sarà ugualmente evocabile con VETTOR + 2. Queste due voci contengono, la prima l'indirizzo della subroutine di servizio PREALL, l'altra quella della subroutine RIPRIST.

Routine INIZIO

Da un confronto con l'esempio precedente, chi legge non avrà alcuna difficoltà a riconoscere le stesse istruzioni qui raggruppate in modo più semplice e, forse, più organico:
modo "0" su porto 5; modo 3 su porto 4; tutti i bit di questo sono di input; viene poi caricato VETTOR, parte "High order" su I e "Low order" sul porto di controllo di (4); segue la parola di controllo interrupt, che qui si è subito pensato di darla con il bit IE (Interrupt Enable) alto, dato che far precedere una disabilitazione è ozioso. I bit di controllo sono anche qui considerati in OR, poi la maschera che segue indica come tali i primi tre (n.b. 1FH = 00011111B). Infine si chiude con IM2 ed EI.

Programma P. 7-2

Label	Cod. Assembly	Commenti
VETTOR	DEFW PREALL	1° vettore di interr.
	DEFW RIPRIST	2° vettore
INIZIO	LD A,0FH	modo 0 su
	OUT (7),A	porto 5
	LD A,FFH	modo 3 su
	OUT (6),A	porto 4
	OUT (6),A	ancora FF su port 4: tutti i bit sono di input
DACAPO	LD HL,VETTOR	carica vettor address sul
	LD A,H	registro I e
	LD I,A	
	LD A,L	sul
	OUT (6),A	porto 4
	LD A,10110111B	carica parola di
	OUT (6),A	controllo interrupt
	LD A,1FH	primi 3 bit
	OUT (6),A	"monitored"
	IM2	

Label	Cod. Assembly	Commenti
	EI	
	RET	
INIMAIN	CALL INIZIO	inizializza porti
	JR 0	resta in attesa
PREALL	IN A,(4)	
	LD C,A	porto 4 in comodo
	CALL RITFISS	100 ÷ 500 msec.
	IN A,(4)	rileggi p. 4
	CP C	come prima?
	JR NZ,FINEINT	se NO, basta con l'interr.
	AND E0H	maschera primi 3 bit
	OUT (5),A	segnalali su p.5
	LD C,A	
	LD A,C0H	maschera bit 7 e 6
	AND C	
	CP C0H	7 = 6 = "1"?
	JP Z,ALLARM	se SI, allarme!
	LD A,60H	maschera i
	AND C	bit 6 e 5
	CP 60H	entrambi = 1?
	JP Z,ALLARM	se SI, allarme!
	LD A,A0H	infine osserva
	AND C	i bit 7 e 5
	CP A0H	e, se tutti e due = 1
	JP Z,ALLARM	all'armi!
FINEINT	XOR A	spegni
	OUT (5),A	luci di allarme
	EI	
	RETI	
ALLARM	....	opportune
	....	istruzioni
	LD HL,VETTOR + 2	punta alla routine RIPRIST
	LD A,L	(°)
	OUT (6),A	
	LD A,10110111	interr. control word
	OUT (6),A	
	LD A,EFH	unico bit di interr. è
	OUT (6),A	il bit 4
	JP FINEINT	(l'incubo è finito)
RIPRIST	....	opportune
	....	istruzioni
	CALL DACAPO	
	RETI	(°°)

(°) - È sufficiente, se si ha l'accortezza di porre i due vettori in indirizzi aventi il byte high order *comune* (es. 0C50 e 0C52), altrimenti occorre anche: LD A,H e LD I,A.
(°°) - Non occorre far precedere la EI, in quanto essa è già prevista in fondo alla routine DACAPO.

Programma MAIN

Anche qui esso è ridotto ai minimi termini e, dopo la chiamata della routine che abbiamo testé descritto, si pone in stato di attesa interrupt.

Routine di servizio PREALL

Al primo interrupt causato dall'azione di uno dei tre sensori connessi alle linee 7, 6 o 5 del porto 4, si entra in PREALL, la cui parte introduttiva è praticamente uguale a quella del caso precedente e serve ad eludere (o, come già detto, fare il "filtraggio software") condizioni di durata inferiore ad un certo minimo, diciamo 100 ÷ 500 msec. Qui RITFISS è una routine che dà un ritardo fisso (e potrebbe essere evitata una CALL, semplicemente inserendo le istruzioni di ritardo).

A questo punto si hanno istruzioni volte a verificare se *almeno due* tra i bit 7, 6 e 5 sono alti. Prima si mascherano i tre bit con l'istruzione AND E0H. Tenendo conto che E0H = 1110000B dovrebbe esser chiaro che, se indichiamo con x, y e z lo stato dei tre, dopo questa istruzione si avrà: xyz00000B nell'Accumulatore.

Questa mascheratura sarebbe stata superflua se ad esempio al porto 5 fossero collegati solo tre LED, ma nelle applicazioni una cosa del genere può essere rara, spesso anzi le linee disponibili non bastano.

Viene caricato in C l'Accumulatore, poi in questo è posto C0H, cioè un valore caratterizzato da tutti i bit nulli tranne il 7 e il 6. L'istruzione AND C produce in A il dato xy000000B, quindi nel successivo CP C0H il flag di uguaglianza Z si accende solo se x = y = 1. In tal caso, si comanda il salto alla label ALLARM.

Si procede in moso del tutto analogo per le coppie di bit 6 e 5, mascherandoli con 60H e con i bit 7 e 5, mascherandoli con A0H. Non si dovrebbero incontrare difficoltà a verificare che, nei due casi, nell'accumulatore si producono i dati 0yz00000 e, rispettivamente, x0z00000. In entrambi, con due "1" si salta ad ALLARM.

Il lettore dotato di logica capisce al volo che così operando anche il caso x = y = z = 1 è implicato, quindi non sfugge all'analisi vista.

Se poi il lettore conosce un poco di Algebra di Boole si renderà pure conto che nelle istruzioni viste si è fatta la simulazione software della funzione:

$$\text{ALLARM} = A_7 \cdot A_6 + A_6 \cdot A_5 + A_7 \cdot A_5$$

risparmiando così un po' di gate esterni.

Si sarebbe potuto anche procedere, con lo Z80 e non con l'8080, in un altro modo, utilizzando le istruzioni di test dei bit. Il flow chart di figura 7-4 è,

riteniamo, sufficiente per illustrare la variante che segue:

```
            BIT 7,A
            JR Z,GIU          se bit 7 = 0 test bit 6
            BIT 6,A
            JP NZ,ALLARM
SU          BIT 5,A
            JP NZ,ALLARM
            JR SEGUITO        (bit 6 = bit 5 = 0)
GIU         BIT 6,A
            JR NZ,SU
SEGUITO     .......           resto del programma
```

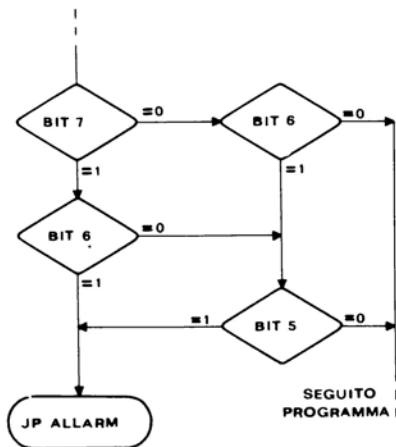



Figura 7-4

Questa soluzione è molto più compatta, anche se meno elegante.

Se l'allarme non merita di essere lanciato, dopo aver spento le luci sul porto 5 (per un caso più generale era forse meglio ri-mascherare i tre bit...) si esce con le solite EI e RETI. A proposito di quest'ultima, ribadiamo che è questa e non la normale RET che deve terminare la routine di servizio con il funzionamento in modo 2 della CPU (rivedere figura 7-1).

Routine ALLARM

Dopo le opportune istruzioni, azionamento sirene, relé ecc. ecc., previste (o simulate dal sullodato volonteroso) all'inizio della ruotine, si carica VETTOR + 2, sul porto di controllo di (4) e, con la successiva parola di controllo (identica alla precedente, ma necessaria di nuovo per introdurre una diversa maschera) si va a precisare che l'unico bit cui ora deve rispondere

l'interrupt è il bit 4. A questo possiamo pensare che sia collegato il pulsante di rimessa in marcia dell'impianto o simili.

A questo punto, con un salto a FINEINT, allo scopo soprattutto di comprendere le istruzioni EI e RETI, si rientra al solito punto di attesa del MAIN, l'istruzione di salto su se stesso JR 0.

Quando infine, il pulsante connesso alla linea 4 del porto 4, o chi per esso, verrà azionato, la PIO cercherà sulla tabella interrupt all'indirizzo VETTOR + 2 e andrà perciò ad eseguire la routine RIPRIST. In questa, dopo istruzioni di inizializzazioni opportune, con la CALL DACAPO va, come è subito evidente, a rimettere in gioco le normali istruzioni di programmazione del porto 4, con il vettore di interrupt di nuovo dato da VETTER, il monitoraggio dei bit 7, 6 e 5 (momentaneamente inibiti) eccetera.

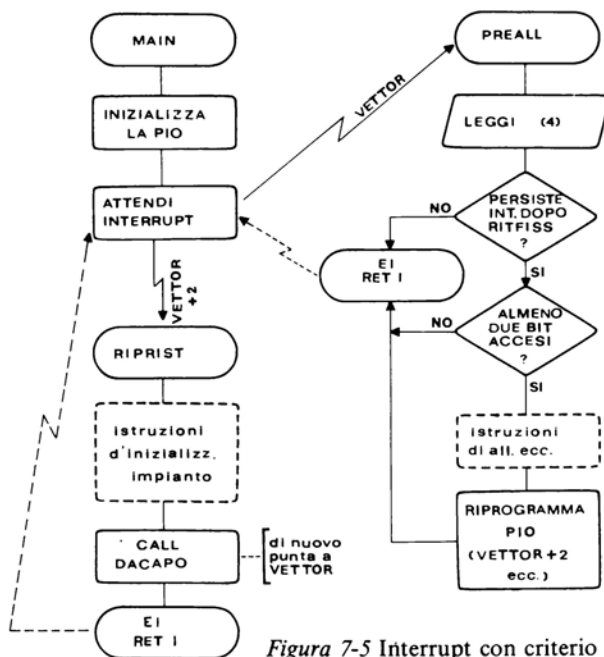



Figura 7-5 Interrupt con criterio di maggioranza (flow chart riepilogativo del P. 7-2).

In casi di questo genere, non c'è niente di meglio che una dettagliata descrizione del sorgente, tuttavia abbiamo ugualmente tentato di riassumere tutti i giri che si sono visti in un flow chart (fig. 7-5). In esso l'interrupt è rappresentato con linee saettanti con sovrapposto l'indirizzo della relativa tabella interrupt.

Speriamo che riesca sufficientemente chiaro.

Programma P. 7-3

STUDIO DELLE PRIORITA' NEGLI INTERRUPT

Questo programma, meno intricato del P. 7-2, è interessante in quanto serve a verificare sperimentalmente le priorità, negli interrupt, tra i due porti A e B della PIO (nel nostro caso si tratta dei già noti 4 e 5): *il primo ha la precedenza sull'altro*. Con queste prove perciò si potrà comprendere in piccolo quello che avviene in una "daisy chain", ossia: quando il dispositivo di priorità più alta interrompe, se è in corso la routine di servizio di un altro si priorità inferiore, questa viene sospesa e riprende solo al termine della ruotine di gestione dell'interrupt del porto più "importante" (ossia: *ubi maior minor cessat*!)

Programma principale (TESTPR)

Stavolta la tabella degli interrupt vettorizzati prevede due elementi, chiamati VETTA e VRTTB, il primo relativo al porto A e il secondo al B.

Queste voci (indirizzi, mettiamo, C50 e C52) contengono rispettivamente gli indirizzi delle routine LAMPO e FULMINE.

L'inizializzazione del modo 3 su entrambi i porti A e B non ha altra particolarità di rilievo che il fatto di avere qui caricato FFH in A una volta sola, in quanto serve in tutte e 4 le istruzioni seguenti (sia per fissare il modo 3 che per indicare tutti i bit come bit di input).

La parola di controllo inviata poi al (6) col suo valore F7H sta ad indicare, come ormai il lettore sarà in grado di capire da solo:

- interrupt abilitato; operatore AND; attivo il livello "H"; segue maschera.

La maschera 3FH, è parimenti facile vedere, fissa come bit di interrupt i primi due. Analogamente si procede per il porto B, per il quale solamente il bit 7 è oggetto di "monitoring" per l'interrupt.

Si hanno poi i consueti caricamenti dei vettori: stavolta, come è ovvio, oltre che VETTA su porto A, si pone subito dopo VETTB sul B.

A questo punto, prima delle EI e RETI, poi di nuovo una delle versioni già viste del ... cane che si morde la coda, sono state messe delle istruzioni specifiche di questo esempio. Queste servono a inizializzare con l'ASCII '5' (hex. 35H) i registri B e C, nonché a puntare, tramite le coppie HL e DE alle posizioni 0BCB e 0BDB della "top row" del video display.

Lo scopo di ciò è chiarito dall'analisi delle routine di servizio.

Routine LAMPO e FULMINE

Chiamate scherzosamente così affinché chi lo vuole possa sostituire ad esse le più barocche subroutine di lampeggi, messaggerie ecc., in realtà sono entrambe volte a fare il conto alla rovescia da 5 fino a 0, poi dacapo, con il numerino che appare nelle già citate posizioni video.

Prima del decremento di B o di C è prevista una pausa di almeno 5 secondi.

Programma P. 7-3

Indir.	Contenuto	Label	Cod. Assembly	Commenti
			ORG D00	
0D00	3A 0D	VETTA	DEFW LAMPO	vettore porto A
D02	48 0D	VETTB	DEFW FULMINE	vettore porto B
D04	3E FF	TESTPR	LD A,FFH	inizio programma; modo 3
D06	D3 06		OUT (6),A	su porto A
D08	D3 06		OUT (6),A	con tutti i bit = input
D0A	D3 07		OUT (7),A	stesso modo 3 e tutti i
D0C	D3 07		OUT (7),A	bit = inp. anche su B
D0E	3E F7		LD A,F7H	parola cntr. con AND, su
D10	D3 06		OUT (6),A	porto A
D12	3E 3F		LD A,3FH	con bit 7 e 6 "monit."
D14	D3 06		OUT (6),A	
D16	3E B7		LD A,B7H	solito contr. word (con OR)
D18	D3 07		OUT (7),A	su porto B, con
D1A	3E 7F		LD A,7FH	solo bit 7 "monitored"
D1C	21 00 0D		LD HL,VETTA	vett. interr. p. A in HL
D1F	7C		LD A,H	parte "High order" nel
D20	ED 47		LD I,A	registro I
D22	7D		LD A,L	e "Low o." su
D23	D3 06		OUT (6),A	porto A
D25	23		INC HL	passa al
D26	23		INC HL	vettore p. B
D27	7D		LD A,L	(°) v. nota
D28	D3 07		OUT (7),A	carica VETTB su p. B
D2A	3E 35		LD A,'5'	ASCII '5' in A
D2C	47		LD B,A	poi in B e
D2D	4F		LD C,A	in C (inizializz. de-cont.)
D2E	21 CB 0B		LD HL,0BCBH	HL punta a sin. "top row"
D31	11 DB 0B		LD DE,0BDBH	DE a destra, stessa riga TV
D34	ED 5E		IM2	n.b. *prima* IM2
D36	FB		EI	*poi* EI
D37	76	STAND	HALT	
D38	12 FD		JR STAND	se torni da int., ri-HALT!
D3A	CD 57 0D	LAMPO	CALL ATTESA	attendi (almeno) 5 sec.
D3D	70		LD (HL),B	visualizza decont. B
D3E	05		DEC B	prepara nuovo valore B
D3F	3E 2F		LD A,2FH	carica in A '0' - 1 (°°)
D41	B8		CP B	vi è giunto B?
D42	20 11		JR NZ,FININT	se NO, va a finire interrupt
D44	06 35		LD B,'5'	se SI di nuovo B = 5 ASCII
D46	10 0b		JR FININT	poi ancora a FININT
D48	CD 57 0D	FULMINE	CALL ATTESA	5 sec.
D4B	79		LD A,C	(il punt. DE lavora solo con A)
D4C	12		LD (DE),A	dec. C su TV
D4C	0D		DEC C	C- 1
D4D	3E 2F		LD A, 2FH	'0' - 1 in Acc.

Indir.	Contenuto	Label	Cod. Assembly	Commenti
D4F	B9		CP C	C è a questo punto?
D50	20 02		JR NZ,FININT	se NO, fine interr.
D52	0E 35		LD C,'5'	se SI ricomincia con C = '5'
D54	ED 4D	FININT	RETI	
D56	D9	ATTESA	EXX	salva puntatori e BC
D57	..	. . . .		istruzioni ad libitum
....	EXX		recupera puntat. e BC	
			RET	fine ATTESA (5 sec. circa)

Il motivo di ciò risiede nel fine dell'esperimento. Questo, dopo il caricamento del programma, consiste nel lanciarlo dando l'execute a partire da TESTPR, poi, a turno si azionano i pulsanti connessi ai bit 7 e 6 del porto A (constatando la funzione AND: solo premendoli insieme l'interruzione funziona) e al bit 7 di B, con una certa lentezza tra una battuta e l'alta. Ogni volta si dovrà constatare il "conto alla rovescia" della cifra visualizzata, rispettivamente, sulla sinistra e sulla destra dello schermo. Quando si arriva a zero, il conteggio riprende da '5', come è facile rendersi conto analizzando la codifica P. 7-3 ed i non scarni commenti a fianco di ogni istruzione segnati.

La codifica in ASCII di 5 risiede nel solito motivo: la ROM generatrice di caratteri richiede tale codice per far comparire 5 poi 4, 3 ecc. È chiaro che questo lavoro può farsi con semplicità solo per cifra massima pari a 9.

Il divertimento comincia ora: si schiaccia prima il pulsante del porto B, poi la coppia di A, senza lasciar troppo tempo in mezzo alle due azioni.

Tutti coloro che sono riusciti a seguire i nostri discorsi sulla priorità degli interrupt non si meraviglieranno di constatare che: *prima* si ha il "count down" relativo al porto A (cifra sulla sinistra dello schermo), *poi* il bravi Z80 in collaborazione con la PIO si ricorda di ultimare il suo compito e scala anche la cifra relativa al porto di priorità inferiore. Infatti, la sospensione di una routine di servizio non implica il suo abbandono definitivo, ma viene ripresa al termine di quella prioritaria. Tutto ciò corrisponde al meccanismo di funzionamento previsto in una daisy chain.

Altre possibilità

Gli esempi che abbiamo visto non esauriscono certo la materia dell'I/O, degli interrupt, delle priorità. Ci sarebbe da parlare di tutte le altre interfacce come la SIO, la CTC ecc. Ci accontentiamo di aver aperto la finestra su un mondo vasto ed affascinante.

Da parte nostra è doveroso avvertire che nei programmini sottoposti alla attenzione le alternative possibili erano molte, per esempio utilizzando altri modi della PIO o anche della CPU. Per esempio, nel P. 7-3, rinunciando alla funzione AND tra i pulsanti 7 e 6 connessi al porto A, si sarebbe potuto molto più semplicemente definire il modo 1 (di Input) per entrambi i porti. Così facendo, è solo necessario inviare la parola di controllo di formato 1XXX011B (X, al solito, è uno stato indifferente). In questo caso, contrariamente a quanto avviene nel modo 3 della PIO, l'interrupt non proviene dalle linee dati del porto (riservate ora, appunto, ai dati veri e propri di ingresso da una periferica esterna) ma dalla linea strobe (ASTB o BSTB), che invece nel modo 3 è inutilizzata (anzi, inibita).

A tale propostito, un esperimento curioso, anche se decisamente assurdo, consiste nel realizzare un orologio (si riveda l'esempio del capitolo 6) collegando al BSTB un punto della linea di divisori di frequenza del clock del NASCOM che fornisce i 50 Hz (serve per il gruppo video display): ad ogni colpo di tale "sincronizzatore" la routine di servizio scatta di una unità, aggiornando i cinquantesimi di secondo, i decimi, i secondi ecc.

E per la funzione di regolazione e/o quella di cronometro? Semplice: basta connettere un pulsante alla linea ASTB del porto A che, avendo priorità sull'altra, ci consente di uscire dalla routine di incremento tempi, entrando in una in cui, tramite la nostra vecchia e cara KBD, potremo momentaneamente bloccare il conteggio, facendo la regolazione tra un cinquantesimo di secondo a l'altro... Non c'è male come rompicapo, ma siamo quasi certi che molte persone ingegnose proveranno a realizzare quest'idea.

Un compitino più facile, nel quale non occorreranno interrupt, ma solamente tre pulsanti connessi al porto A e sei LED sul B, il primo come input e il secondo come output, consiste nel rifare la simulazione del semaforo in modo più realistico. La visualizzazione sullo schermo si può anche lasciare, immaginando un monitoraggio dello stato del crocicchio. Sul MAIN si dovranno allora aggiungere solo le istruzioni per l'accensione dei LED giusti del porto B (°). Lasciando al lettore questo compito abbastanza semplice, suggeriamo qui la parte iniziale della routine RITPOL, conservandone la stessa struttura illustrata nel flow di Fig. 6-1. Si confronti anche, naturalmente, con la codifica del programma P. 6-1. Si ha:

```
RITPOL   XOR A          per fare Cy = 0
INPUT    CALL RITFIX
         IN A,(4)       lettura/polling del porto input
         OR A           attiva i flag
         JR NZ,GIU      se A diverso da 0, va a vedere GIU
         DJNZ INPUT     decr. B e torna a INPUT con B = 0
```

(°) - C'è quasi da arrossire a dire che, dei sei LED, tre rappresentano i colori di una via, gli altri tre quelli dell'altra. E ancor più a suggerire l'impiego di LED diversamente colorati.

```
GIU     SCF         metti in "on" il Carry flag
        BIT 5,A     è il comando "lampeggio"?
        RET Z       se NO (bit 5 = 0!) rientra, con Cy = 1
LAMP    ......      (resto del programma da modificare)
```

Osservazioni

La XOR A serve a fare Cy = 0 come, nel P. 6-1 la AND A (che andava ovviamente bene anche qui, come OR A). La OR A dopo la operazione IN A,(4) serve ad attivare i flag (che la operazione IN lascia invece immutati) in conformità al valore messo in Accumulatore, onde la successiva JR NZ,GIU operi correttamente, quando A ≠ 0. Saltando a GIU si accende il Cy, onde segnalare avvenuto azionamento pulsante e si fa il test del bit 5 (corrispondente al comando "lampeggio"): se il bit 5 è "off", cioè flag Z = 1, si rientra, altrimenti si prosegue con la parte LAMP eccetera (rivedere sempre, bene, il P. 6-1). Nel MAIN poi, in luogo di SW si potrà utilizzare, a piacere, lo stato del bit 7 o del bit 6 (stanno al posto di 'V' e 'R', rispettivamente).

Due sole cose dobbiamo infine rimarcare: a) si è qui per semplicità ignorato ogni problema di *debouncing*, che sarà meglio affrontato in capitoli successivi; b) è sottinteso che, se i bit diversi da 7, 6 e 5 del porto 4 sono inutilizzati, essi sarà opportuno vengano ancorati a massa, affinché diano con sicurezza sempre zero. Qualora si utilizzassero per altri scopi bit 4,3 ecc. potrebbe invece risultare necessaria una "mascheratura" dei bit 7, 6 e 5, ponendo a zero tutti gli altri, il che si farebbe nel modo seguente (queste istruzioni seguono la IN A,(4)):

AND E0H seguita da JR NZ,GIU

infatti ora la OR A non occorre più, dato che la AND E0H (cioè 11100000B) attiva il flag Z in conformità al valore zero/non zero di A.

Tornando infine al programma P. 7-2, il lettore sottile si sarà accorto di un inconveniente: nella ipotesi di un persistere dell'attivazione di un solo sensore (guasto, ad esempio), si continua ad uscire dal MAIN, anche se senza effetti "catastrofici", (grazie al filtro software pro...maggioranza silenziosa. Chiaramente, benché nel caso in questione il fatto non dia alcuna conseguenza, altrettanto non può dirsi in un MAIN più "serio" che faccia altro che girarsi i pollici. Il rimedio, che al solito lasciamo a chi studia, potrebbe consistere nella disattivazione del port (tramite l'invio della parola di controllo di formato IE XXX0011, con il bit IE =) e l'emissione di un segnale di avviso guasto sensore (perché si provveda alla riparazione): qui può tornare utile l'idea di ripristino insita nel programma P. 7-2 medesimo. Va anche detto che questo è un caso tipico in cui è preferibile non esser troppo taccagni e utilizzare dell'hardware esterno: ad esempio con tre semplici porte NAND si può ricavare la funzione booleana: AB + BC + AC, dove A, B e C sono gli stati dei tre sensori incriminati.

"..... e come è duro calle lo scendere e 'l salir..."
(Dante, Paradiso)

CAPITOLO 8

CONTROLLO DI UN ASCENSORE (SIMULAZIONE)

Parleremo ora del controllo di un impianto che, essendo a tutti familiare, si presta molto bene ad essere compreso nelle sue specifiche essenziali: l'ascensore.

Mettiamo subito le mani avanti: il criterio che ci ha guidato è in definitiva didattico, ossia far apprendere le possibilità offerte dall'uso di un microprocessore nelle sue più tipiche applicazioni. In questo ambito, e dati anche i limiti... fisici dell'opera, non è possibile esaurire un problema in tutti i suoi aspetti. Questo anzi appesantirebbe molto il discorso, oltre che rubare spazio ad altri esempi, per una panoramica un po' più completa.

Con le semplificazioni che è allora giocoforza introdurre, nel nostro ascensore didattico mancano funzioni concettualmente secondarie ma dal punto di vista pratico essenziali come il comando per l'apertura e la chiusura delle porte, l'azionamento di due velocità del motore (una più bassa per la partenza e l'arrivo al piano), le sicurezze ecc. Molte di tali funzioni si potrebbero aggiungere ai programmini sviluppati, altre si possono immaginare affidate all'hardware esterno. Quest'ultima impostazione però non è di norma la più usata, in quanto il microprocessore tende ad essere un accentratore, a causa delle sue grandi possibilità: solo così, ossia col risparmio di componenti esterni, si può oltretutto giustificare la spesa, a volte non indifferente, di sviluppo del progetto a micro, specialmente per quanto si riferisce al software ed ai suoi tempi un po' lunghi.

È infine doveroso dire che non siamo perfettamente sicuri che gli ascensori operino esattamente secondo le modalità che andremo ad ipotizzare; poco male: questo non è il trattato del perfetto ascensorista e se i nostri pochi ma maliziosi nemici vorranno declassare i nostri impiantini a montecarichi si accomodino pure. Parlando più seriamente, in una materia come questa non è infrequente che le specifiche varino notevolmente da un caso all'altro e da un cliente all'altro, onde non è detto che specifiche "inventate" non siano proprio quelle che qualcuno richiede.

Cominciamo in questo capitolo con esempi di simulazione sul nostro sistemino di sviluppo.

Programma P. 8-1

ASCENSORE SENZA PRENOTAZIONE (simulazione)

Come tutti i casi di simulazione già visti, si userà la tastiera sulla quale si possono battere i tasti da 0 a 9 (il tasto "0" potrebbe significare piano terra) e la unità video nelle posizioni, per esempio, 08DA e 08DE, delle quali la prima destinata a contenere uno dei simboli seguenti:

'H' per indicare stato di "HALT"
'↑' per indicare "SALITA"
'↓' per indicare "DISCESA"

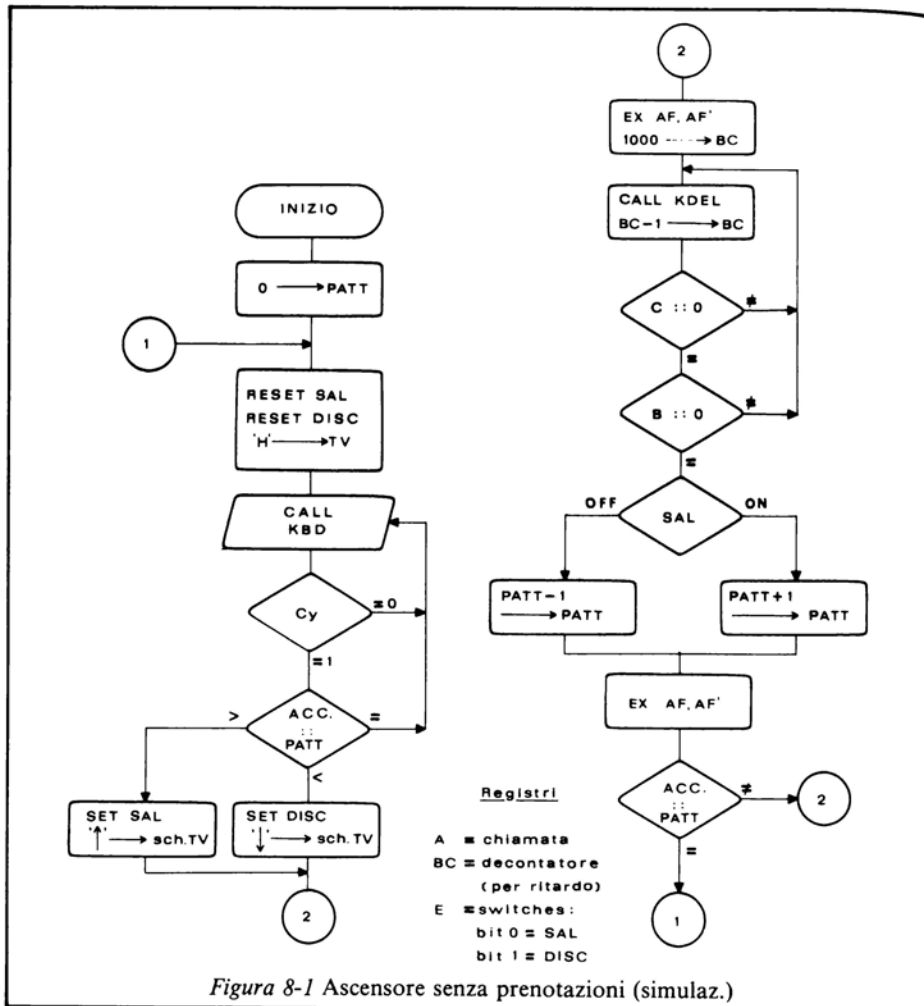



Figura 8-1 Ascensore senza prenotazioni (simulaz.)

Il generatore di caratteri già varie volte citato è in grado di produrre questi ed altri simboli particolari (previsti per applicazioni tipo "video games", ma anche per cose più "serie", come la schematizzazione di impianti ecc.).

La seconda è utilizzata per indicare il *piano attuale*, PATT e funziona sia come contatore up/down (cioè reversibile) che come dispositivo di display del piano via via raggiunto.

Si fa notare che abbiamo scelto la limitazione a 9 piani + p.t. in quanto così facendo il codice ASCII generato dalla keyboard è direttamente compatibile con quello della citata ROM. Con più di nove piani, oltre alla complicazione di rappresentare due cifre sul TV monitor, ci sarebbe stata quella anche maggiore derivante dal dover decodificare altri tasti (es. A, B, C ecc.) per rappresentare pulsanti al di sopra di 9. Tutte complicazioni assolutamente fuori luogo in una simulazione didattica.

Flow chart

Come si può vedere nella figura 8-1, si inizia ponendo zero nel campo PATT e resettando i flag indicati simbolicamente con SAL e DISC, di significato immediato, costituiti in pratica dallo stato dei bit 0 e 1 del registro E.

Questi due bit hanno un duplice scopo: oltre ad indicare la condizione, meglio lo *stato*, "salita" e "discesa" della cabina, al tempo stesso simulano i comandi verso il gruppo motore (per semplicità estrema si pensa ad una sola velocità) o meglio verso i relé che controllano l'arresto, la marcia in un senso e quella in senso contrario.

Si pone quindi 'H' sullo schermo: insomma stiamo simulando una partenza dal piano terra. A questo punto interviene (indovinate chi?) la consueta routine del monitor KBD, con l'altrettanto solito loop d'attesa (salto relativo di -3 e richiamata di KBD se $Cy = 0$). Se il tasto è premuto, $Cy = 1$, si va al confronto tra l'Accumulatore, che contiene il valore del piano chiamato e il contenuto del campo PATT (°): se i due dati concordano, si tratta evidentemente di errore e, dato che comunque sul piano chiamato ci siamo già, si torna all'attesa di una nuova chiamata...più seria; se invece il contenuto di A è maggiore di PATT, viene messo in "on" lo switch SAL e la freccia in alto è fatta comparire sullo schermo; viceversa, con la condizione opposta è il deviatoio DISC ad essere attivato e la freccia rivolta in basso messa sul video, là dove finora era 'H'.

A questo punto, la marcia dell'ascensore è simulata semplicemente con

---

(°) - Avvertiamo per l'ennesima volta, che nei flow chart con un nome simbolico si usa indicare il *contenuto* di un'area quale che sia. In Assembly invece la label è associata ad un indirizzo e, se si vuole il contenuto di quell'area di memoria, occorrono parentesi alla label.

un ritardo di circa 5 secondi per ogni piano (si è scelto un tempo un po' breve solo per evitare la noia dell'attesa).

Questi 5 secondi si sono ottenuti al solito modo, inserendo in un loop KDEL dopo aver posto in principio il valore 1000 nella coppia BC, quindi contando a ritroso fino a che C = 0 e pure B = 0, altrimenti si va avanti con il conteggio...indietro.

Dopo tale tempo, si interroga SAL e se esso è ON, si aumenta di un'unità PATT, altrimenti lo si decrementa. In tal modo, ogni 5 o più secondi (se si imposta un dato più realistico), si vede indicato sullo schermo il nuovo piano raggiunto. L'istruzione EX AF,AF' in testa e la medesima in coda alle istruzioni che abbiamo or ora descritto servono, la prima, a porre in salvo (sui registri "fantasma") l'altra a ricuperare il valore dell'Accumulatore contenente la chiamata ultimamente fatta. Questo perché, come si vedrà meglio nella codifica, il gruppo di istruzioni di ritardo utilizza l'Accumulatore.

A questo punto, si torna a paragonare A con PATT e, se diversi, si torna ad attendere altri 5 secondi (connettore 2), se invece il piano raggiunto PATT uguaglia quello chiamato, tramite il connettore 1 si ricomincia con il reset dei due deviatori-azionatori di relé per l'arresto del motore e si segnala lo stato di Halt con la comparsa di 'H' sul video. Il ciclo continua poi con un'altra eventuale chiamata, a volontà.

Codifica

Dopo il flow chart e le dettagliate spiegazioni fornite, il programma in Assembly non presenta difficoltà e il lettore che ci ha seguiti bene fin qui potrebbe tranquillamente saltare queste note.

Se ci dilunghiamo, lo facciamo solo a vantaggio dei lettori più pigri e a titolo di incoraggiamento per coloro che hanno avuto difficoltà nell'infernale capitolo che precede.

Viene inizialmente messo l'ASCII 30H in PATT, ossia si fa comparire lo zero in questa posizione video. Si poteva anche fare: LD A,'0' poi LD (PATT),A, per un eguale numero di byte impiegati.

Un'osservazione merita la riga di indirizzo D15 e le seguenti, che qui riportiamo per comodità, insieme alle due istruzioni che precedono:

```
LD HL,PATT
CP (HL)
LD HL,CARATT
JR Z,CHIAMA
JR P,GIU ecc.
```

L'istruzione LD HL, CARATT, che punta alla locazione video ove compaiono il simbolo 'H' o le frecce, è stata messa in questa posizione per evitare di doverla scrivere due volte.

Il puntatore è preparato in anticipo in modo che, quando gli indirizzi D1F e D25 si hanno le istruzioni volte, rispettivamente, a caricare in CARATT la freccia "in giù" o quella in su, si punti alla medesima posizione

08DA della video RAM. Il lettore pignolo può provare a verificare quel che sarebbe successo tralasciando questa istruzione anticipata, riscrivendo que-

Programma P. 8-1

Indir.	Contenuto	Label	Cod. Assembly	Commenti
		KBD	EQU 69H	
		KDEL	EQU 35H	
		CARATT	EQU 08DAH	
		PATT	EQU 08DEH	
			ORG D00H	
0D00	21 D3 08	ASCENS	LD HL,PATT	
D03	36 30		LD (HL), '0'	zero ASCII, su TV, PATT
D05	1E 00	AGAIN	LD E,0	reset bit 0 e bit 1 reg. E
D07	21 DA 08		LD HL,CARATT	
D0A	36 48		LD (HL),'H'	'H' su TV (posiz. CARATT)
D0C	CD 60 00	CHIAMA	CALL KBD	attendi chiamata
D0F	30 FB		JR NC,CHIAMA	
D11	21 DE 08		LD HL,PATT	
D14	BE		CP (HL)	cfr. Acc. con (PATT)
D15	21 DA 08		LD HL,CARATT	predisponi puntatore
D18	28 F2		JR Z,CHIAMA	A = (PATT): non muov.
D1A	F2 23 0D		JP P,GIU	A magg.? va GIU
D1D	CB CB		SET 1,E	A minore: switch DISC "on"
D1F	36 0B		LD (HL),0BH	0BH = freccia giù, su TV
D21	18 04		JR ALTPIAN	altro piano...
D23	CB 03	GIU	SET 0,E	A magg.: set SAL
D25	36 5E		LD (HL),5EH	5EH = frecc. su, in TV
D27	08	ALTPIAN	EX AF,AF'	metti in salvo chiamata
D28	01 E8 03		LD BC,1000	inizio ritardo
D2B	CD 35 00	ATTESA	CALL KDEL	di 1000 x 5 msec. = 5 sec.
D2E	0B		DEC BC	
D2F	79		LD A,C	
D30	B7		OR A	per attivare flag Z
D31	20 F8		JR NZ,ATTESA	
D33	B8		CP B	
D34	20 F5		JR NZ,ATTESA	
D36	21 DE 08		LD HL,PATT	dopo 5 sec. prepara puntat.
D39	CB 43		BIT 0,E	salita o discesa?
D3B	20 03		JR NZ,SOT	salita: va SOT
D3D	35		DEC (HL)	disc.: diminuisci un piano
D3E	10 01		JR PIUSOT	
D40	34	SOT	INC (HL)	sal.: aumenta piano
D41	08	PIUSOT	EX AF,AF'	ripristina in A chiamata
D42	BE		CP (HL)	cifr. chiam. con nuovoo
D43	CA 05 0D		JP Z,AGAIN	siamo arrivati!
D46	18 DF		JR ALTPIAN	c'è ancora un piano (almeno)

sto tratto di programma: l'istruzione LD HL,CARATT va in questo caso inserita due volte, la prima a monte dell'indirizzo in cui v'è l'istruzione LD LD (HL),0BH, la seconda prima dell'istruzione LD (HL),5EH.

Qualcuno si chiederà: ma non c'è qualche rischio a lasciare in sospeso il risultato della comparazione CP (HL), sita all'indirizzo D14? la risposta è *nagativa*: come si può verificare dall'Appendice B questa istruzione, come quasi tutte quelle di "loading" non altera alcun flag, quindi Z e S risultano quelli conseguenti la precedente comparazione CP (HL).

Un identico criterio si è impiegato alla riga D36: qui per maggior chiarezza l'istruzione di puntamento anticipato LD HL,PATT è stata scritta prima del test BIT 0,E. In tutto si sono così risparmiati 3 x 2 = 6 byte.

Quanto alle istruzioni relative all'ottenimento del ritardo vogliamo solo rilevare una particolarità. Dopo il decremento di BC (indirizzo D2E) si è qui pensato di caricare C in A, onde verificare se C è o meno uguale a zero. Attenzione però: non si può fare subito dopo JR NZ,ATTESA, proprio per il motivo or ora visto che il caricamento non è un'operazione che influenza alcun flag. È allora necessario inserire l'istruzione OR A, la quale non modifica in alcun modo il valore caricato nell'Accumulatore, ma attiva i flag in conformità a tale valore.

Era peraltro più semplice fare, come già visto in altre occasioni:

...... DEC BC poi XOR A CP (HL) ecc.

e si risparmia anche un byte.

A proposito infine di risparmio di byte, dato che qui non si hanno problemi di numeri relativi si può porre in luogo di JP P,GIU (indirizzo D1A) una JR NC,GIU interrogando il flag Carry, anziché quello del segno S.

Programma P. 8-2

ASCENSORE CON UNA SOLA PRENOTAZIONE (simulazione)

Si vuole aggiungere la possibilità di fare, dopo la prima chiamata, una prenotazione. La cosa appare utile, sia che si pensi ad una chiamata esterna, mentre la cabina è in viaggio, sia che ci si riferisca al caso di più di una persona che si trovi entro la cabina. Qui introduciamo almeno due semplificazioni: la prima già presente nel programma precedente consiste nel non fare distinzione tra chiamate esterne ed interne; la seconda invece sta nel limitare ad una soltanto il numero delle prenotazioni. A proposito della prima limitazione essa senz'altro non è del tutto realistica, basti pensare alla comparsa dell'avviso "In arrivo" che ha evidentemente senso solo per chiamate esterne, tuttavia, a nostro modesto parere la distinzione non è concettualmente essenziale e il significato basilare dell'ordine è, in entrambi i casi, "raggiungere il tal piano".

Circa la seconda semplificazione essa ha chiaramente il significato di primo approccio al problema delle prenotazioni multiple, anche qui però la personale ed abbastanza allucinante esperienza di un ascensore "intelligente" (che continuava, per "eccesso di zelo" a salire e scendere, alla Figaro-qua-Figaro-là) ci suggerisce l'idea che una limitazione del numero delle prenotazioni possa persino essere salutare (meglio far aspettare qualcuno che portare altri continuamente a spasso...).

Bando comunque ad ulteriori divagazioni, altrimenti rischiamo di scrivere un trattato "De Ascensoribus" in più volumi.

Flow chart (Figura 8-2)

Allo scopo di memorizzare la prenotazione si utilizza il registro D, nel diagramma a blocchi indicato simbolicamente con il nome PMEM. Viene anche introdotto un terzo deviatore, oltre ai soliti SAL e DISC già utilizzati in precedenza. Lo ciameremo, banalmente, SW e utilizzeremo a tale fine il bit 2 del medesimo registro E di prima. Questo bit serve a ricordare, con il suo stato ON, l'avvenuta prenotazione ed è interrogato due volte nel corso del programma:

- all'arrivo ad un piano;
- durante, o poco prima della corsa.

La parte introduttiva è identica al caso illustrato nel diagramma di Fig. 8-1, con l'aggiunta della messa in OFF del nuovo deviatore SW. Si esegue poi un sottoprogramma di ritardo (circa 1,3 secondi) denominato PAUSA, la cui funzione non è bene sia anticipata ora, la vedremo tra non molto.

A questo punto, prima di entrare nel solito loop di attesa di una chiamata di keyboard, viene interrogato il deviatore SW. Questo naturalmente all'inizio è spento, giusto quanto abbiamo fatto all'entrata del programma e in tal caso si prosegue nello stesso modo del caso precedente.

In seguito, tuttavia, se una prenotazione è avvenuta, SW potrà essere acceso. In tale caso, dopo aver messo il piano prenotato PMEM nell'Accumulatore e, ancor prima dopo aver messo "off" SW, si by-passa il ciclo di attesa di una nuova chiamata, in quanto ce n'è già una ancora in attesa di essere soddisfatta, e si salta al confronto tra registro A e PATT (si noti come con questa impostazione la prenotazione abbia la precedenza su un'altra chiamata). Qui possiamo allora anche comprendere il perché dell'introduzione del sottoprogramma PAUSA dopo l'halt: se questo intervallo di tempo non ci fosse, nel caso di arrivo ad un piano con il debito di una precedente prenotazione si avrebbe sì la comparsa della lettera 'H', ma questa subito scomparirebbe (e il nostro occhio imperfetto non si accorgerebbe di nulla). Facendo così, simuliamo anche tempi realmente necessari per operazioni

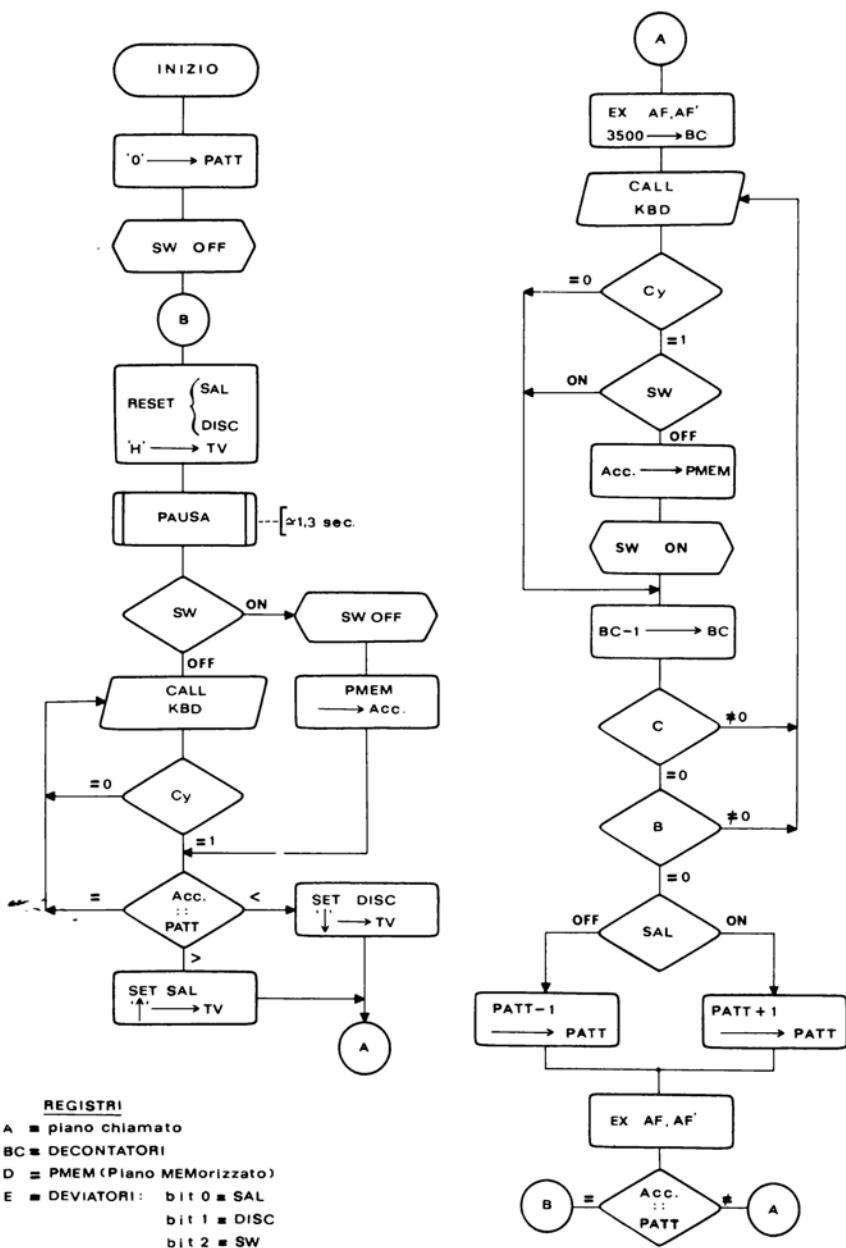



Figura 8-2 Ascensore con prenotazione (prima versione imperfetta)

come l'assestamento della cabina, l'apertura delle porte et similia (tutte cose che noi, con la più tranquilla impudenza, abbiamo deciso di ignorare...).

Passando poi alla parte che simula la marcia verso il piano, si è aggiunta ora una CALL KBD, per così dire "aperta", in cui cioè anche nella consizione Cy = 0 non torna su se stessa ma va avanti a decrementare la coppia di registri BC eccetera. In tal modo si fa ugualmente il "polling" della keyboard, ma inserendo nel ciclo che opera il ritardo simulante il tempo di passaggio da un piano all'altro. Non è più necessario introdurre la chiamata di KDEL, in quanto si è pensato di sfruttare il ritardo di circa 1,5 msec. che la KBD fornisce quando nessun tasto è premuto; per ottenere ancora 5 secondi, si inizializza BC a 3500.

A cosa serva il polling della tastiera nell'ambito dei loop di ritardo di 5 secondi è abbastanza evidente: stare all'erta per una eventuale chiamata durante la corsa o, come l'abbiamo chiamata, una prenotazione. Ciò si verifica naturalmente con la condizione del Carry flag acceso. In questa evenienza si interroga anche il deviatore SW: se esso è spento, ciò vuol dire che nessuna altra prenotazione è stata fatta nel frattempo, pertanto si va a porre il dato, proveniente da tastiera ora esistente in Accumulatore, sul registro D, ossia PMEM. Subito dopo SW viene attivato. In tal modo, ad una successiva interrogazione di SW questo devierà il programma in modo da fargli saltare le istruzioni dianzi discusse, andando direttamente al punto in cui si decrementa BC. Tutto questo perché, come si è detto, abbiamo deciso in partenza di limitare le prenotazioni ad una solamente.

Si noti infine come qui la funzione dell'istruzione EX AF,AF' risulti, per così dire, ancora più essenziale di prima, dato che l'Accumulatore serve anche per accogliere una seconda eventuale chiamata.

Flow chart migliorato

Se si riflette un poco, si può però osservare che la versione appena vista ha una imperfezione. Supponiamo infatti che si stia andando dal primo verso il 6° piano (per una chiamata per così dire diretta o per soddisfare una precedente prenotazione) e si faccia una prenotazione per il 4° piano: è evidentemente un vero peccato passarvi davanti senza fermarsi!

La miglioria che abbiamo pensato è molto semplice (e prima di andare a sbirciare la Fig. 8-3 dove essa è illustrata il lettore accorto provi a trovare una propria soluzione e poi a confrontarla con la nostra).

Al raggiungimento di ogni piano, cioè, con la nostra simulazione, allo scadere dei detti 5 (o più) secondi, si fa come prima il confronto innanzitutto di PATT con il dato, or ora rimesso con la EX AF,AF' in Accumulatore.

Subito dopo però c'è la "coda" che corrisponde alla nostra miglioria.

Si va a vedere se lo switch SW è acceso o meno. Se è spento, ossia non è pendente alcuna prenotazione, si torna attraverso il connettore A all'inizio del sottoprogramma di simulazione, passaggio (di nuovo in su o in giù *come*

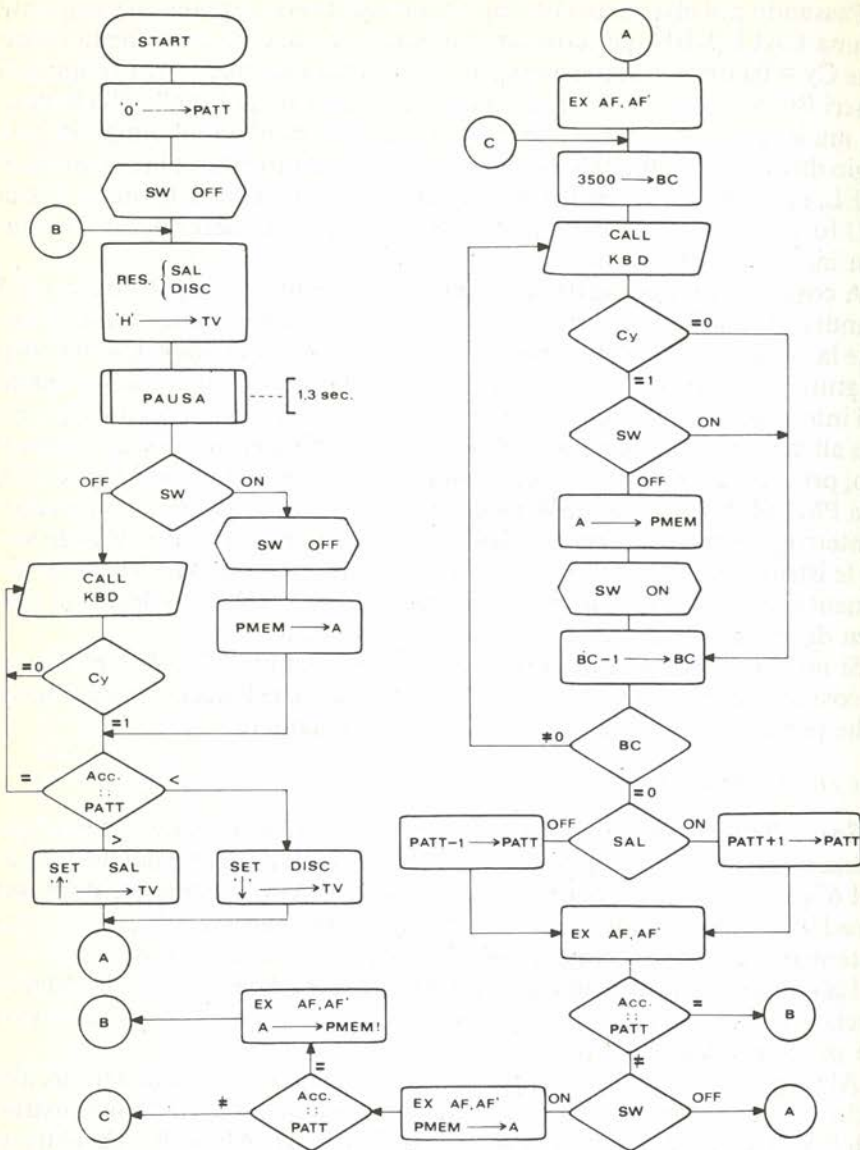



Figura 8-3 Ascensore con prenotazione (migliorato)

prima) ad altro piano (label ALTPIAN della precedente codifica).

Se invece esso è in "on", si va a confrontare PATT con PMEM e, se c'è uguaglianza si torna alla esecuzione della fermata.

Nella realizzazione pratica di questo principio ci si può trovare facilmente in un "impasse" e solo dopo vari tentativi si trova la soluzione corretta. Innanzitutto appare evidente che, per poter eseguire il confronto tra il piano prenotato e quello attuale è necessario mettere PMEM in Accumulatore ma, prima ancora, è necessario operare il salvataggio in A' dell'Accumulatore, altrimenti la chiamata diciamo così "regolare" che era stata testé messa in A andrebbe perduta. Uscendo ora dal confronto Acc. con PATT, cioè in sostanza tra prenotazione e PATT, se si ha la condizione di diversità ci si deve collegare, connettore C, all'istruzione immediatamente dopo la

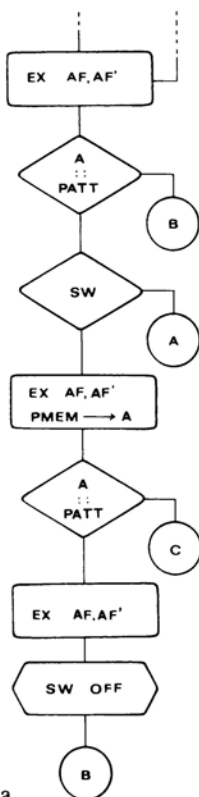



Figura 8-4 Soluzione errata.

prima di quella destinare a simulare la ripartenza (in altri termini: non alla label ALTPIAN, ma ad ALTPIAN + 1). Questo perché ora l'istruzione EX AF,AF' è già stata eseguita.

Con la condizione invece di uguaglianza tra piano prenotato e attuale, si rischia abbastanza facilmente di compiere l'errore illustrato nella Fig. 8-4: collegarsi al punto B in cui comincia la fermata dopo aver spento il deviatore SW (dato che la prenotazione sta per essere soddisfatta) e richiamare di nuovo A' in A. Ma se si va a fare la verifica di questo provvedimento si trova subito che, al rientro in B, dopo la simulazione della fermata più la routine PAUSA, con la condizione off di SW si incontra la CALL KBD "chiusa su se stessa" e, quindi, ci si ferma senza che la chiamata, che pure era stata fatta per prima, venga soddisfatta! Viene allora in mente di provare a non spegnere SW. In questo caso, dopo le dette operazioni di fermata simulata, SW fa deviare il programma sulla destra, ma ora è PMEM ad essere messo nell'Accumulatore, così poi avviene che il successivo confronto dà condizione di uguaglianza e di nuovo... si sta fermi.

Abbiamo voluto insistere nell'illustrazione di queste difficoltà in cui è non improbabile che si sia trovato il lettore che non ha barato (ed ha provato per suo conto, seguendo il nostro invito, ad elaborare una sua soluzione, prima di andare a guardare la Fig. 8-3). Da questa discussione si dovrebbe anche comprendere quale è la soluzione del rompicapo: essa consiste nel mettere la primitiva chiamata in PMEM (registro, ricordiamolo, D). Ciò viene fatto, come si vede in Fig. 8-3 prima richiamando A dal registro "occulto" A' poi ponendo A in PMEM.

Il provvedimento è logico e solo in apparenza cervellotico: una volta arrivati al piano prenotato, visto che abbiamo deciso che questo passaggio non va "sciupato" *i ruoli tra chiamata primitiva e prenotazione si invertono.*

A questo punto si dovrebbe confidare nella bontà della soluzioni adottata dal momento che essa è stata giustificata logicamente. Tuttavia, per evitare brutte sorprese in seguito, cioè dopo che è stata compiuta, magari un poco penosamente, la codifica, è meglio non vergognarsi a fare quel "*debug passo passo sulla carta*" che in altre occasioni abbiamo suggerito.

Vediamolo. Supponiamo che si faccia inizialmente (partendo dal piano terra) la chiamata del 4° piano e, mentre si viaggia dal 2° al 3°, sia prenotato il sesto. Proviamo a descrivere a parole i successivi eventi. Chi legge è opportuno che si aiuti con una tabella in cui segnerà i successivi valori dei vari registri e flag.

Si parte con SW off, poi si esce dalla CALL KBD con Cy = 1 e, dal confronto seguente, con Acc. maggiore di PATT, per cui si ha il set dei SAL ecc.

Inizia il loop che simula la salita da un piano all'altro. Al primo trascorrere dei già detti 5 secondi, si esce sempre da CALL KBD con Cy = 0 e si decrementa BC fino a che non è zero. A questo punto PATT è accresciuto di un'unità. Dal successivo confronto della chiamata (4° p.) con PATT (che ora vale 1) si esce con *diverso* e, poiché ancora SW è off, si ritorna tramite il

connettore A ad un altro intervallo di 5 secondi e la storia si ripete identica anche al passaggio al 2° piano, per quanto detto. Dopo di che, durante uno dei giri di "ritardo + polling della tastiera, viene battuto il tasto 6. Stavolta si esce da KBD con Cy = 1 ed essendo SW off la prenotazione è accettata e messa in PMEM, mentre SW va in on. Continua poi il decremento di BC. Quando si torna al valore BC = 0, PATT passa a 3, il confronto con la prima chiamata dà diversità, ma stavolta, essendo *on* SW, si confronta PATT anche con PMEM che dà pure diversità ecc. Finalmente, arrivando PATT a 4, si esce, nel confronto Acc. con PATT, dal connettore B, viene eseguita la simulazione di fermata, con pausa. Ora si incontra SW *on*, il programma devia sulla destra, PMEM, ossia 6 è messo in Accumulatore e il confronto successivo di nuovo accende SAL e scrive la freccia volta in alto sullo schermo. Lo switch SW, dimenticavamo di dire, era stato spento. Per farla breve, ora supponiamo che nella marcia verso il 5° piano venga prenotato, appunto, il 5°. Vediamo la situazione all'arrivo al 5° piano, cioè, sul flow chart subito a monte del confronto Acc./PATT (dopo la EX AF,AF'):

A	A'	D	PATT	SAL	DISC	SW	
6	-	5	5	on	off	on	e, dopo i successivi due confronti e: EX AF,AF'; D in A:
5	6	5	5	on	off	on	dopo A cfr. PATT e: EX AF,AF'
6	5	6	5	on	off	on	seguito da A trasf. in D:

quindi si esce con il connettore B, si esegue la "fermata" al piano 5 e la pausa, si incontra SW che devia a destra, SW è messo *off*, PMEM ossia D, nell'Accumulatore e si ha: A = 6 e D, invece...pure! Questa situazione a prima vista paradossale, nonché il fatto di trasferire in A, da D, un valore che *c'è già*, non deve meravigliare. Né ciò ci deve indurre a pensare che nessuna delle due istruzioni che precedono l'ultima uscita dal connettore B (EX AF,AF' e D trasferito in A) sia superflua (provare a toglierne una dopo l'altra, per credere). La piccola istruzione oziosa che, *in questo caso*, trasferisce un valore già esistente è la conseguenza del fatto che in precedenza si era fatto il trasferimento opposto (ed i trasferimenti, come si sa, conservano il dato del registro emittente). Comunque, il lettore se ne convinca, è questa la soluzione più semplice, e ciò che più conta, essa come si è verificato (il resto in sostanza è stato già detto...) *funziona.*

Codifica (in Assembly)

Lasciando al lettore la cura di tradurre in linguaggio assoluto, riportiamo di seguito il programma in linguaggio Assembly. Dopo le abbondanti spiegazioni precedenti, non sarà difficile, sia confrontando con l'esempio precedente comprendere tutte le peripezie ivi contenute.

Facciamo qui solamente alcune osservazioni “tecniche”.

Innanzitutto le istruzioni comprese tra la riga 11 e la riga 15 comprese, ossia da CALL NZ,SWOFF a quella etichettata con la label CFRPATT: il funzionamento è chiaro, quando SW è “on” si esegue la routine SWOFF, riportata nel fondo del programma, in cui sono comprese le sole istruzioni di spegnimento di SW e di caricamento in A della prenotazione D. Altrimenti si prosegue; subito dopo, sulla condizione NZ, si salta a CFRPATT (cioè confronto Acc. con PATT). Così, nel caso in cui Z = 0, né si fanno le operazioni della routine SWOFF, né si salta la CALL KBD e seguenti. Soluzione abbastanza elegante, che, come alternativa “scolastica” avrebbe:

```
          JR NZ,AVANTI
ATTKEY    CALL KBD
          JR NC,ATTKEY
          JR CFRPATT
AVANTI    RES 2,0
          LD A,D
CFRPATT   LD HL,PATT
```

Se si va a fare il conto dei byte, in questo ultimo caso ne occorrono 15, contro i 17 della soluzione adottata. La piccola differenza (che in altri casi consimili, a conti fatti risulta addirittura a favore della CALL condizionata) è compensata, a nostro avviso, dalla maggior organicità e chiarezza.

Un’altra nota la meritano le istruzioni RES 0,E e RES 1,E: rispetto al programma precedente, si è dovuto rinunciare all’azzeramento dell’intero registro E degli switch, perché ciò ora avrebbe comportato la messa in off anche del nuovo flag SW, che invece, come è chiaro dal flow chart, non va resettato al rientro alla label FERMATA (connettore B, sul flow).

Da ultimo, si osserverà una piccola differenza tra la precedente versione e l’attuale: il puntatore HL è predisposto all’indirizzo PATT subito a monte della label ALTPIAN, mentre nell’esempio di prima veniva fatto tutte le volte che BC diventava zero. È una finezza, starei per dire una “gentilezza” nei confronti della Z80-CPU per risparmiargli un po’ di fatica, dato che nel corso di tutta la parte successiva ad ALTPIAN il valore di HL non muta più (ritornerà a CARATT, con la FERMATA).

Problema delle prenotazioni multiple

Vogliamo ora generalizzare al caso di quattro (o più, questo numero serve solo per fissare le idee e, per dieci piani, è più che sufficiente).

Definiamo, anche in base alle riflessioni che possono scaturire dagli esempi finora visti, i *criteri di servizio:*

1) l’ascensore parte in salita o in discesa *in conformità alla prima chiamata;*

2) durante la corsa accetta prenotazioni e le soddisfa ogni volta che, *man-*

Programma P. 8-2

Label	Cod. Assembly	Commenti
	LD HL,PATT	inizio programma
	LD (HL),'0'	'0' su (PATT)
	RES 2,E	SW = off
FERMATA	RES 0,E	reset SAL
	RES 1,E	e DISC
	LD HL,CARAT1	
	LD (HL),'H'	'H' su TV
PAUSA	LD B,0	rit.do di circa
	CALL KDEL	1,3 secondi
	DJNZ PAUSA	
	BIT 2,E	com'è SW?
	CALL NZ,SWOFF	se SW = 1, SW off e D in Acc.
	JR NZ,CFRPATT	se SW = va al cfr. con PATT
ATTKEY	CALL KBD	aspetta caratt. (key)
	JR NC,ATTKEY	
CFRPATT	LD HL,PATT	punta con HL su PATT
	CP (HL)	cfr. Acc. con (PATT)
	JR Z,ATTKEY	ignora chiamata fasulla
	JR NC,GIU	se A > PATT va GIU
	SET 1,E	set DISC
	LD (HL),0BH	freccia in giù
	JR ALTPIAN	altro piano
GIU	SET 0,E	set SAL
	LD (HL),5EH	freccia in su!
	LD HL,PATT	punta su PATT
ALTPIAN	EX AF,AF'	
	LD BC,3500	(rit. di 1,5 x 3500 msec ≅ 5 sec.)
ATTPREN	CALL KBD	"polling" della prenotaz.
	JR NC,DECBC	se non c'è va al decr. di BC
	BIT 2,E	che fa SW?
	JR NZ,DECBC	se è "on" va a DECBC
	LD D,A	Acc. in PMEM
	RES 2,E	SW off
DEC BC	DEC BC	solita tecnica per
	XOR A	fare BC − 1
	CP B	ecc. finché BC = 0
	JR NZ,ATTPREN	
	CP C	
	JR NZ,ATTPREN	
	BIT 0,E	con BC = 0 osserva SAL
	JR NZ,SOT	se SAL = 1 va SOT
	DEC (HL)	discesa: decrem. (PATT)
	JR SEGUITO	poi va al SEGUITO progr.
SOT	INC (HL)	salita: incr. (PATT)
SEGUITO	EX AF,AF'	recupera chiamata
	CP (HL)	cfr. con (PATT)

Label	Cod. Assembly	Commenti
	JR Z,FERMATA	se uguali, va a FERMATA
	BIT 2,E	e il buon SW che dice?
	JR Z,ALTIPIAN	è spento? altro piano
	EX AF,AF'	SW = on: metti in salvo chiam.
	LD A,D	PMEM in Acc.
	CP (HL)	cfr. con (PATT)
	JR NZ,ALTPIAN + 1	se diversi, dacapo, saltando la EX AF,AF'
	EX AF,AF'	ripesca chiamata
	LD D,A	considerala prenotazione!
	JP FERMATA	
SWOFF	RES 2,E	SW off
	LD A,D	PMEM in Acc.
	RET	fine routine (e END progr.)

tenendo la corsa nello stesso senso, passa davanti ad uno qualsiasi dei piani prenotati;

Questo criterio è più semplice, ma a nostro avviso più logico, in una applicazione di questo genere: nell'ascensore troppo "intelligente" di cui parlavamo l'inversione di piano per soddisfare una prenotazione proveniente da un piano vicino suscitava smarrimento e ira tra gli occupanti la cabina!

3) una volta arrivati al piano chiamato per primo se ci sono ancora prenotazioni, viene soddisfatta quella effettuata per prima in ordine di tempo: criterio che, oltre che rispondere ad un evidente principio di giustizia, consente, se del caso, l'inversione della marcia.

Una volta fissati gli obiettivi, o, come dicono i tecnici, *le specifiche* del problema, la stesura del flow chart non presenta (eccessive) difficoltà. Esso è illustrato nella Fig. 8-5. Dalla sua analisi potremo constatare che nelle grandi linee esso ricalca lo stesso della Fig. 8-3, anche se alcuni dettagli sono stati "assorbiti" entro sottoprogrammi, sia per motivi si spazio e compattezza, sia per dare una maggior generalità possibile al tutto. Tuttavia, per meglio fissare le idee, conviene ugualmente che si tratti della stessa simulazione sul video già vista.

La prima e forse più importante novità consiste nell'uso predominante del doppio registro HL come puntatore alla *tabella delle prenotazioni.*

Poiché gli altri registri hanno lo stesso significato di prima e cioè: A = chiamata; E = flag vari (SAL, DISC e SW); BC = decontatore per simulare tempo di passaggio da un piano all'altro (e/o puntatore alla video RAM, V. più avanti);

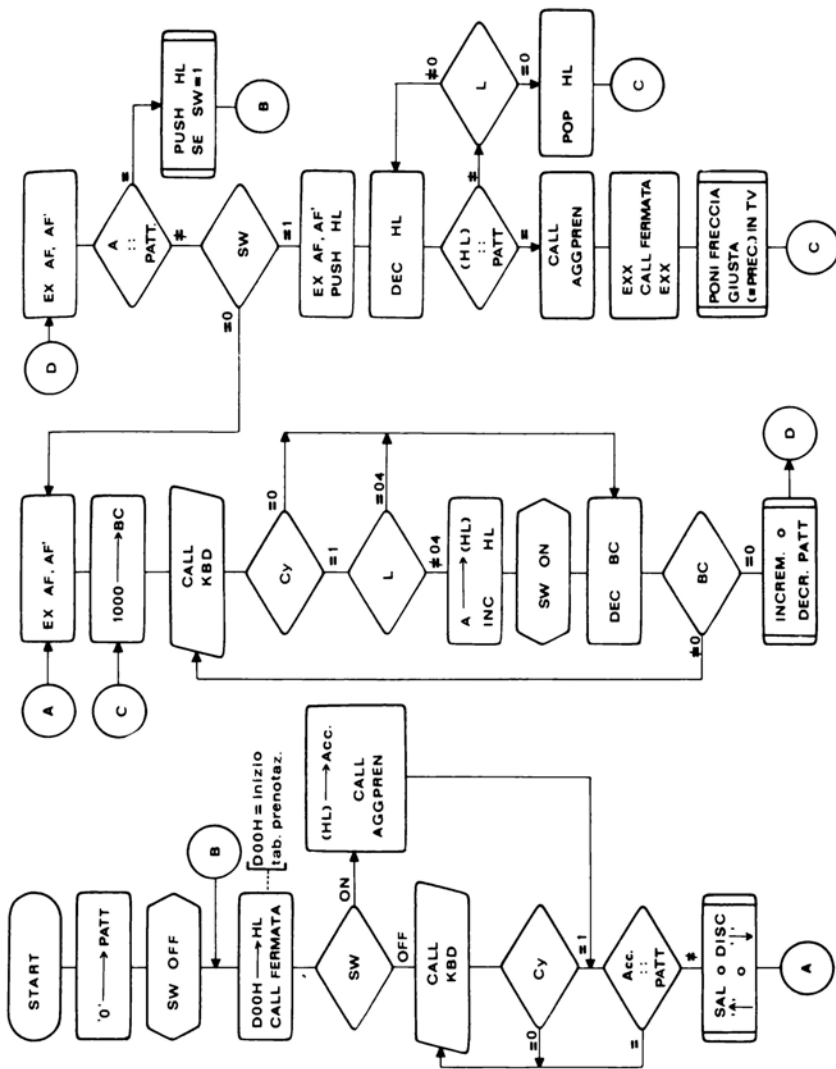



Figura 8-5 Ascensore con più prenotazioni.

per puntare alle posizioni della video RAM che chiamavamo PATT e CARATT (°) dovrà essere fatta un po' di attenzione ed arrangiarsi.

La soluzione più banale, anche se un po' noiosa e tale da allungare alquanto il sorgente, è di far precedere PUSH HL e seguire POP HL ogni volta che si deve puntare a dette posizioni video salvando il valore di HL relativo alla tabella prenotazioni (questo ad es. non occorre con le prime istruzioni con le quali si pone '0' in PATT). Un'altra possibilità che suggeriamo a chi volesse tradurre in linguaggio Assembly consiste nello sfruttare il registro D, ora libero, per depositare la chiamata. Si potrà così utilizzare come puntatore a PATT e CARATT la coppia BC, ad es. così:

LD A, CARATT poi LD BC,08DAH e LD (BC),A (°°)

Naturalmente, così facendo il confronto di A con PATT andrà sostituito con quello tra D e PATT.

Descrizione dettagliata del flow-chart

Il lettore confronti costantemente con la Fig. 8-3. Si inizia allo stesso modo poi, al connettore B, si incontra l'istruzione di posizionamento del puntatore HL alla posizione D00H. Questa è la prima della tabella delle prenotazioni, per la quale sarà opportuno riservare *sette* locazioni di memoria (da D00 a D06), per motivi che saranno visti tra un momento.

Segue poi la chiamata di una subroutine FERMATA, che corrisponde alla PAUSA, al reset di SAL e DISC ecc. del diagramma di prima, ma potrà immaginarsi comprendente qualunque altro gruppo di istruzioni del genere.

Si è qui preferita la soluzione di una CALL per maggior chiarezza dell'insieme ed anche perché a voler impostare le cose come nella coda di Fig. 8-3 ci sarebbero state complicazioni facilmente intiuibili. FERMATA è invocata anche nella parte di flow in cui si soddisfa una prenotazione "en passant". Così facendo si risparmiano alcune istruzioni.

Tutte le altre istruzioni dei blocchi sulla sinistra, fino al connettore A sono identiche al caso precedente, tranne quelle sulla destra del deviatore SW trovato in "on": si ha l'istruzione di trasferimento di (HL) in Accumulatore, seguita dalla chiamata della subroutine AGGPREN la quale, secondo modalità che vedremo più avanti, provvede a togliere dalla detta tabella la prima prenotazione che ora viene considerata chiamata prioritaria; si fa così anche posto ad un'altra prenotazione.

(°) - Magari intervallati di due soli spazi (es. 08DA e 08DC) in modo da poter passare dall'uno all'altro con due INC BC o due DEC BC.

(°°) - N.B. non esiste l'istruzione LD (BC),CARATT. Per questo fa comodo avere disponibile l'Accumulatore.

Il primo dei due blocchi che abbiamo appena detto è del tutto analogo all'istruzione PMEM trasferito in A di Fig. 8-3. La mancanza dello spegnimento di SW deriva dal fatto che *ora non si ha una sola prenotazione*. Pertanto SW andrà messo in off in una sede più opportuna.

Il resto del diagramma è stato disposto su due file affiancata per motivi di spazio. Ne deriva un più abbondante impiego di connettori, che possono creare un certo ... giramento di testa iniziale. Con un poco di pazienza guariremo anche da questa (lieve) indisposizione.

Partiamo dal connettore A: andando in giù fino al punto in cui la coppia BC è trovata essere uguale a zero, si trova la stessa struttura e la maggior parte delle istruzioni identiche a quelle del vecchio flow. Al posto però dell'interrogazione di SW, c'è ora l'esame del contenuto del registro L, che è la parte di ordine basso della coppia puntatrice (è sufficiente l'esame del solo L, dato che H contiene permanentemente il valore 0D): se si ha L = 04, ciò indica che il registro delle prenotazioni è "saturo" e non se ne accettano più. In caso contrario, si hanno le azioni: a) la prenotazione è caricata sulla locazione (HL); b) con INC HL si passa alla cella successiva, posto eventuale di un'altra prenotazione; c) si accende SW per indicare che una prenotazione (almeno) è stata fatta.

Due osservazioni volanti: 1) la saturazione con L = 04 si ha quando, caricata la quarta prenotazione *nella cella D03*, la INC HL produce appunto L = 04; 2) la continua riaccensione di SW con prenotazioni successive alla prima non disturba affatto il nostro flemmatico Z80!

Quando BC = 0, dopo il sottoprogramma che racchiude le istruzioni di incremento o decremento, a seconda del valore di SAL e DISC, del piano PATT, tramite il connettore D si va, come prima all'EX AF,AF' poi al confronto di A con PATT (°). Sulla condizione di uguaglianza si salta al punto B: si è già visto che, in tale evenienza, prima si esegue la FERMATA, poi, se è SW = 1, l'assunzione della prenotazione nella cella iniziale della tabella di prenotazione nel registro A di chiamata prioritaria.

Prima di proseguire però, si provvede a mettere in salvo sullo stack il contenuto di HL, *ma* (N.B.!) *solo se SW = 1* (abbiamo per brevità racchiuso in un solo sottoprogramma le istruzioni, facilmente immaginabili, di interrogazione SW, con l'esecuzione o meno di PUSH HL). Questa complicazione non ci sembra evitabile: il salvataggio è necessario per la successiva possi-

---

(°) - Coloro che seguissero il suggerimento dato di porre la chiamata in D, non solo, come è pacifico, dovranno operare tale confronto di PATT con D, ma potranno fare a meno delle (noiose) istruzioni EX AF,AF' di salvataggio e recupero della chiamata in A.

bile esecuzione di AGGPREN entro la quale si fa poi il debito POP HL, ma non si deve dimenticare che, con SW = 0 il connettore B può *non* portare a tale situazione ed in tal caso, se non avessimo operato con la prudenza detta, in mancanza di prenotazioni si avrebbe un gonfiamento pericolosissimo dello stack!

Se l'uguaglianza A = PATT non si verifica, anche qui, con SW successivamente pescato in "off" si torna ad altro piano, tramite connettore A.

È a questo punto che le differenze si fanno più marcate. Dopo EX AF,AF' si opera il PUSH di HL, in tal modo si salva l'indirizzo dell'ultima cella si memoria, predisposta per una successiva prenotazione, o quello D04 che, come si è visto, corrisponde alla condizione di saturazione prenotazioni. Dopo aver ricordato che il PUSH fa restare in HL tale valore, il confronto seguente con PATT è (attenzione!) *preceduto dal decremento di HL, puntando così all'ultima prenotazione effettuata.*

Non appaia questa scelta motivata da una bizzarra applicazione della massima evangelica sugli ultimi che diventano primi. Si fa così solo per comodità (altrimenti, partendo da D00, oltre all'istruzione di riposizionamento a questo valore di HL, si sarebbe dovuto tenere la contabilità del numero delle prenotazioni, il che è una complicazione non indifferente. (°)

Dal confronto del contenuto di (HL) con PATT si può uscire con condizione di uguaglianza (e vedremo che succede in tal caso) altrimenti, se L è non nullo, si torna a decrementare HL e a comparare un'altra prenotazione con il piano raggiunto (tutto questo febbrile lavoro di ricerca avviene sul filo dei microsecondi e pertanto, niente paura, la cabina avrà un tempo enorme per arrestarsi).

Se si arriva ad L = 0 ciò ha il significato di "nessuno-vuol-scendere-qui" e, tramite il connettore E, si torna ad un'altra simulazione di passaggio a nuovo piano.

Se invece nella tabella si trova una prenotazione da soddisfare, si invoca la già citata subroutine AGGPREN per l'accomodamento nella tabella, poi si esegue la FERMATA. Questa altra CALL è qui preceduta e seguita dalla istruzione EXX avente lo scopo di mettere al sicuro i flag del registro E (soprattutto SAL e DISC). Questi infatti all'inizio di FERMATA vanno resettati, inoltre deve comparire 'H' sul video (v. la Fig. 8-3). Ora noi abbiamo precisato che il nostro ascensore *non deve invertire rotta fin quando non si è raggiunto il piano prenotato per primo.*

All'uscita di FERMATA è poi necessario sostituire 'H' con la freccia giusta (su o giù): i flag SAL e DISC, ripristinati dalla EXX vengono perciò

(°) - Per lo stesso motivo, non si è ritenuta qui vantaggiosa la istruzione di ricerca tabellare CPDR, o CPIR, tanto utile peraltro in casi consimili.

subito utilizzati: si interroga SAL e a seconda del suo valore *on* o *off* verrà fatta comparire nuovamente la freccia si prima. Come vedete, nel programma bisogna proprio pensarle tutte!

Soddisfatta questa prenotazione, si torna al connettore E.

Subroutine AGGPREN

Data la sua relativa delicatezza, proponiamo una versione, tra le tante possibili, in linguaggio Assembly.

La situazione comune ad entrambi i casi di chiamata di AGGPREN, grazie alla nostra previdenza (modestia a parte) è la seguente:

- *sullo stack*: indirizzo locazione per prenotazione futura;
- *in HL*: indirizzo prenotazione accettata (o che si sta per accettare)

In queste condizioni decidiamo per un algoritmo relativamente rozzo, ma semplice, consistente, come si è accennato all'inizio nello spostare tre celle di memoria a sinistra, *in tutti i casi*, tanto HL nel corso del main, come si è visto, punta sempre, tranne che in fase di ricerca tabellare, alla locazione più alta. L'inconveniente di dover sacrificare altre tre celle sulla destra appare davvero modesto. Comunque di questi spostamenti di tabelle, anche nel modo più completo e sofisticato abbiamo imparato l'uso ai tempi dei fatui giochi delle scritte viaggianti: visto che a qualcosa sono serviti?

La soluzione, con i debiti commenti abbondanti è la seguente:

```
AGGPREN   LD BC,3      prepara lunghezza blocco
          LD D,H       rendi DE = indirizzo cella da
          LD E,L       ricoprire, relativa a prenotaz. soddisf.
          INC HL       HL = indirizzo cella accanto
          LDIR         scorrimento a sinistra del blocco
          POP HL       ricupera vecchio indirizzo di successiva
                       prenotazione
          DEC HL       diminuiscilo di uno
          XOR A
          CP L         compara L con zero
          RET NZ       rientra se ancora prenotazioni
          RES 2,E      altrimenti metti "off" SW
          RET
```

La LDIR opera come segue: sia ad es. HL = D02 e DE = D01; viene messo (D02) in (D01); poi (D03) in (D02) ecc.

Il decremento di HL prima della RET serve a posizionare correttamente l'indirizzo della locazione su cui andrà caricata l'eventuale nuova prenotazione.

A questo punto scopriamo anche dov'è che il deviatoio SW viene rimesso in "off": se la *nuova* prenotazione va messa in 0D00, ossia L = 0, vuol dire che non ce n'è più di pendenti e, in tal caso SW è resettato.

Il cerchio si chiude e chi legge può ripetere tutto il giro per raccapezzarsi meglio (e magari trovare qualche erroruccio, non garantiamo sulla assoluta perfezione di tutto).

A conclusione di questo programma vogliamo ribadire che esso non ha alcuna pretesa di essere un modello. Altre soluzioni e anche impostazioni piuttosto diverse da questa sono possibili. Chi legge potrà divertirsi con esse o, quanto meno a mettere in atto varianti più o meno divertenti come ad esempio la comparsa della scritta "in arrivo" un po' prima dell'alt, oppure far comparire i numeri dei piani in posizioni mobili verticalmente e via simulando.

Tabelle di servizio

Il precedente programma ci offre lo spunto per parlare di una materia abbastanza tipica di problemi di code, gestione traffico e simili. Intendiamo parlare di *tabelle di prenotazione* in cui sono contenute cioè richieste da parte di una o anche più periferiche (impianti o utenti diversi) che sul momento non era possibile soddisfare. Quando si verifica una situazione favorevole alla soddisfazione di un servizio, il microprocessore va a vedere (gli inglesi dicono fare il "*look-up*" della tabella) su tale insieme di dati in cui le richieste sono via via andate accumulandosi.

Il caso che abbiamo visto era, tutto sommato, abbastanza atipico. I casi più correnti sono relativamente più semplici e classificabili, almeno in generale. Per quanto si riferisce al caricamento, di solito si provvede col criterio di porle in posizioni di memoria successive. A volte può essere richiesto da quel particolare problema un ordinamento. Questo potrebbe essere ad esempio fatto nel modo visto nel programma P. 4-2.

Ma la distinzione più corrente riguarda tabelle non preliminarmente ordinate, in modo che l'ultima cella contenga l'ultima richiesta.

A seconda del criterio di accesso a tali insiemi o come anche si dice, "memorie", si distinguono principalmente il tipo LIFO (che vuol dire: *Last In First Out*) e quelle di tipo FIFO (ossia *First In First Out).*

L'esempio più classico del primo genere è ben noto agli utenti di microprocessori: è lo *stack* che, nello Z80, utilizza il registro SP come puntatore solitamente automatico. L'accesso per scrittura è fatto tramite istruzioni PUSH, mentre il POP comanda la lettura. I dati qui sono costituiti da voci, a 16 bit. Questo meccanismo è per molti versi assai pratico (è qui doveroso citare i calcoli a catena che, anziché tramite parentesi, si fanno con il metodo detto RPN = "Reverse Polish Notation" in onore al ricercatore polacco che l'ha inventato e che gli utenti di calcolatori Hewlett Packard ben conoscono).

Tuttavia nel caso di cui stiamo parlando questo uso non ci sembra troppo pratico, e non tanto per i 16 bit, anziché 8, che sono in gioco quanto perché la catasta (stack) è denominata così perché su di essa si ammonticchia di tutto, nelle operazioni di salvataggio: registri e indirizzi. È di solito meglio non rinunciare a questo mezzo potente e crearsi una tabella LIFO separata.

Una tecnica per chi preferisse disporre ancora di istruzioni PUSH e POP potrebbe essere quella di prevedere uno stack separato. A tal fine occorrerà opportunamente manovrare con le istruzioni: LD SP,HL (o le analoghe relative ai registri IX e IY) o LD SP,nn o anche LD SP,(nn), molto utile quest'ultima quando si voglia dinamicamente passare dall'*ultima* posizione di uno stack all'ultima di un altro (°).

Si può anche, per così dire, *simulare* uno stack, ad esempio mediante il registro doppio HL. Da principio HL è fissato all'inizio della tabella (che, contrariamente a quanto capita con l'usuale stack, preferiamo disporre secondo indirizzi crescenti) e viene fatto "fluttuare" avanti con istruzioni INC HL seguita da LD (HL),r (r = A, B ecc.) oppure LD (HL),n ossia dato immediato; mentre il passaggio indietro impiegherà istruzioni tipo LD r,(HL) seguita da DEC HL. È chiaro che nei due casi si imita un PUSH e, rispettivamente, un POP (di dati a 8, anziché a 16 bit).

Tabelle FIFO. Un'implementazione chiara e sistematica potrebbe essere fatta impiegando due puntatori, ad esempio HL e DE. I due vengono posti inizialmente all'indirizzo più basso della tabella. Per scrivere, dopo il loading del dato in (HL), si incrementa HL, mentre per leggere si preleva tramite DE e quindi si incrementa quest'ultimo. Può anche essere opportuno conservare in qualche registro o area di memoria il dato *lunghezza* (o dimensione) dell'insieme: posto inizialmente a zero ed incrementato insieme ad HL, è comodo per sapere in ogni momento se, ad esempio, la tabella è ancora vuota. La dimensione può comunque essere anche ricavata per differenza tra il contenuto di HL e quello di DE.

La memoria che abbiamo descritto funziona in un modo che implica, ad ogni successiva lettura con DE, l'automatica indifferenza per le celle "già servite": da questo punto di vista essa si presenta di dimensioni massime *illimitate* un po' come lo stack. A differenza di quest'ultimo però qui *si va sempre avanti* il che costituisce un evidente pericolo.

(°) - Il monitor NASBUG usa appunto un piccolo stack riservato, con salvataggio in memoria RAM del valore dell'user stack (e ripristino, al rientro).

Per ovviare ad esso vi sono due possibilità:

a) imporre un *limite massimo* per HL, al superamento del quale, se ancora nessun servizio è stato effettuato non si accetta la nuova prenotazione (°) mentre, quando almeno un servizio è stato effettuato non si accetta la nuova prenotazione (°) mentre, quando almeno un servizio è stato soddisfatto si rilocano tutte le prenotazioni a partire dall'inizio. Le istruzioni atte a tale fine sono le seguenti (per fissare le idee si pensino i seguenti valori indicativi - con HL, non si dimentichi che punta ad una locazione predisposta per una *futura* prenotazione) HL = 101; DE = 06F e INIZ = 050:

```
XOR A          per il reset Cy, dato che c'è solo SBC tra HL e
               DE;
SBC HL,DE      calcola lungh. blocco: (101-6F)H = 92H, in HL
LD B,H         lunghezza blocco
LD C,L         in BC (Byte Counter nella LDIR)
EX DE,HL       DE in HL: HL = 6FH, cioè iniz. blocco "old"
LD DE,INIZ     DE = 50H, INIZio blocco "new"
LDIR           copia "old" in "new"
```

A questo punto, la situazione, coi dati dell'esempio, è la seguente: BC = 0; DE = (50 + 91) H = E1H e HL = 100); occorre ora aggiustare i corretti valori dei puntatori DE e HL con le istruzioni:

```
EX DE,HL       HL punta all'ultima prenotaz. sita in E1H;
INC HL         HL punta a nuova prenotazione
LD DE,INIZ     DE punta a prima prenotaz. inevasa, ora in 50H;
```

b) una tecnica analoga di rilocazione tabella si può sistematicamente fare *ogni volta che una prenotazone è servita*: anche se si richiedono spostamenti di blocchi più frequenti (con conseguente rallentamento dell'elaborazione) questa soluzione ha più pregi: minor impiego di memoria; soprattutto si può anche eliminare, come puntatore sistematico, DE, dato che ora la posizione di prelievo delle prenotazioni è fissa alla cella iniziale.

La routine di rilocazione blocco, con suo scorrimento a sinistra di un posto è stata già vista in altre occasioni, ma ne riportiamo ugualmente una ver-

---

(°) - O la si pone in qualche area di riserva (es. lo stack) segnalando al tempo stesso la situazione di "overflow" all'esterno. A volte però è il problema stesso ad imporre un limite al numero delle prenotazioni.

sione per comodità di chi segue (con lievi varianti):

```
PUSH HL          salva puntatore a futura prenotazione
XOR A            Cy = 0 (non si sa mai)
LD DE,INIZ       (nell'ipotesi di abolire DE come punt. fisso)
SBC HL,DE        calcolo lunghezza blocco
PUSH HL          lungh. blocco sullo stack e,
POP BC           dallo stack in BC (olé!)
LD HL,INIZ + 1
LDIR
POP HL           recupera puntatore HL
DEC HL           diminuiscilo di uno (°)
```

Si noti la finezza del PUSH HL seguito da POP BC come mezzo alternativo alle due istruzioni (vista nel caso di prima): LD B,H e LD C,L (non ci si distragga poi mai, in casi come questi a invocare istruzioni inesistenti come LD BC,HL o EX DE,BC!; anche se è vero che l'assemblatore si rifiuta di tradurre simili cose turche - o noi stessi ce ne accorgiamo, se creiamo il programma oggetto...artigianalmente - si tratta pur sempre di una perdita di tempo).

A conclusione del capitolo vediamo un ultimo programma di simulazione di un impianto che ha una certa parentela con l'ascensore, anche se viaggia in orizzontale.

Programma P. 8-3

SIMULAZIONE DI UN CARRO PONTE

Anziché un carro ponte si può immaginare qualunque altro dispositivo mosso da un motore avanti e indietro tra una stazione iniziale ed una terminale.

Supponiamo anche vi siano stazioni intermedie in una (solamente) delle quali è prevista una sosta per compiere una determinata operazione (carico o scarico nel caso del carro ponte). L'apparecchiatura, in condizioni di *marcia* (comandate con il tasto 'M' della keyboard) deve continuamente muoversi dalla stazione 0 alla stazione 9, qui simulate dalle posizioni da 0BC0 fino alla 0BC9 della top row (°°), quindi invertire la marcia e così via, in moto

---

(°) - Dato che un servizio è stato reso, anche il posto per successiva prenotazione va spostato a sinistra. *Si noti che, dato che al termine della LDIR, HL punta a fine tabella più uno, PUSH HL, POP HL e DEC HL si possono togliere.*

(°°) - L'uso della top row non è una fissazione: è l'unica che siamo sicuri essere "pulita" all'inizio di un programma, dato che in fase di caricamento essa non è interessata.

alternativo, fino a che non si abbia il comando 'R', cioè *ritorno* alla posizione di partenza. Tra una stazione e l'altra il viaggio del carro ponte è simulato con un ritardo di 1,7 sec. (dato che si tratta di simulazione preferiamo un tempo breve, che ci consente meno istruzioni) mentre all'arrivo ad una stazione si accende il numero corrispondente nella detta posizione video (es. il numero 7 nella 0B17) e vi resta per i già detti 1,7 secondi. Non prevediamo, per estrema semplicità, un comando di arresto (senza ritorno). In definitiva, si potranno dare da tastiera, oltre ai detti 'M' e 'R', anche comandi con i tasti da 0 a 9 che equivalgono all'ordine di una fermata non appena si passa nella corrispondente stazione.

Una volta per tutte facciamo notare che in questi programmini non abbiamo messo istruzioni di rigetto di tasti diversi da tutti quelli previsti. Se nell'ambiente in cui si vive abbondano i soliti pierini, occorrerà armarsi di pazienza ed inserire tali noiosi controlli (e magari chiamate di sottoprogrammi che scrivono opportuni improperi in tali evenienze).

Uso registri: registro **C** = numero ASCII della stazione: evolve da 30H a 39H; *registro* **D** = stazione di fermata: viene riempito con il valore della cifra battuta, anche se la precedente richiesta non è stata esaudita (criterio semplificato ma forse realistico pensando ad un operatore che abbia così la immediata possibilità di annullare una battuta errata); *registro* **B**: solito decontatore per ritardi; *registro* **E**: bit 7 usato come flag AV che simula il comando di "marcia AVanti" e bit 6 come switch imitante "marcia INDietro".

Dal punto di vista strettamente software, sarebbe bastato un solo deviatore, ma il problema avrebbe avuto una minor generalità (il caso AV = IND = 0 simulante l'arresto non è rappresentabile con un bit soltanto). Abbiamo poi qui omesso indicazioni dello stato, tipo 'H' e frecce volte a destra e a sinistra; chi volesse questo complemento è ovviamente liberissimo di farlo, in analogia con l'esempio ascensoristico.

A questo punto, non resta che illustrare il diagramma a blocchi, disegnato nella Fig. 8-6.

Con la messa in off di AV e IND, il puntamento di HL a INIZ (cioè all'indirizzo detto, 0BC0), il caricamento in C dell'immediato 30H, cioè '0' ASCII, ha inizio il programma. Ci sono poi le istruzioni: C trasferito in (HL) che mette in (0BC0) il valore '0', facendolo comparire sullo schermo e il caricamento in D del valore "inesistente" FFH (qualunque altro dato, purché diverso da 30 ÷ 39 andava bene). La funzione di questa particolare inizializzazione del registro D (è un trucco di programmazione abbastanza classico) sarà vista nel corso del programma.

Figura 8-6 Simulazione di un dispositivo tipo carro ponte.

Segue la solita CALL KBD evocata con loop d'attesa. Con Cy = 1, se non si tratta del comando 'M' (o 'R' e non ci sono in giro i "pierini") si considera la battuta come prenotazione della corrisposndente stazione, il cui numero è posto in D, poi si torna comunque ad attendere un altro tasto. Ovviamente, come si vede, la battuta di 'R' è ignorata.

Quando il tasto pigiato è 'M' si inizia la simulazione della marcia con l'ormai familiare tecnica del ritardo-polling preceduta dal set del flag AV. Dopo l'azzeramento del decontatore B (per ottenere, tra una stazione e l'altra, un tempo di circa 256 x 6,5 msec. = 1,7 sec.) viene confrontato il contenuto di (HL) con quello di D: questo nell'ipotesi che anche la stazione iniziale sia di servizio (il confronto poteva anche essere fatto con C) (°). Se i due dati coincidono, viene messo il valore "inesistente" in D, poi si esegue un sottoprogramma, i cui dettagli illustreremo in sede di codifica, ma che provvede a far lampeggiare per una dozzina di secondi la stazione "servita", sottolineando che lì si lavora.

A questo punto si può chiarire il modo di operare di quell'FF: quando non c'è prenotazione, il successivo confronto darà diversità e si va all'istruzione seguente (ed alla seguente stazione) senza lampeggi di sorta. È un modo come un altro per risparmiare un flag.

Quando, nel ciclo di "polling" (che qui incorpora anche KDEL) un tasto è premuto, Cy = 1, se nell'Accumulatore è trovata una 'R' si va alla parte di RITORNO che vedremo da ultima, altrimenti, se è una chiamata erronea 'M' è ignorata (ovvio, siamo già in marcia), in caso contrario si mette nel deposito D la nuova prenotazione.

All'uscita del ciclo che simula transito da una stazione all'altra, si va a interrogare il deviatore AV, dopo aver messo il blank nella posizione finora puntata da HL: se AV è *on* si incrementano C e HL, altrimenti entrambe vengono decrementate, quindi il nuovo valore di C è posto in (HL). Poi si guarda il nuovo valore di C (stazione cui si è appena arrivati): se esso è rispettivamente '0' o '9', si fa il set di AV e il reset di IND o, viceversa, il reset di AV ed il set di IND, lasciando altrimenti le cose come stanno.

Dopo aver sistemato così le cose (senso di marcia e nuova stazione), tramite il connettore X si fa ritorno al ciclo di ritardo/scansione tastiera.

Parte terminale (label RITORNO)

Quando nella scansione della keyboard è individuato un tasto 'R', tramite il connettore Y si salta al reset di AV e set di IND, come è ovvio, poi si fa subito il confronto di C con '0', dato che se per caso ci si trova proprio in

---

(°) - Si noterà però che C non si può abolire, dato che la posizione di memoria visualizzata è ora *mobile* sullo schermo.

corrispondenza della prima stazione possiamo far finta di essere arrivati all'obiettivo. In questo caso resettiamo sia AV che IND, per l'arresto del motore. Qualora invece questa ipotesi non eccessivamente probabile non si verifica, si chiama una routine di ritardo di circa 0,30 secondi, trascorsi i quali si decrementano C e HL, dopo aver messo naturalmente in (HL) il blank, poi si mette nel nuovo (HL) il nuovo C ecc. finché C = 0 al che si fa quanto già visto sopra: reset AV e IND ecc.

Se si riflette bene, in questa impostazione si riscontrano due rozzezze: la prima, forse più evidente è che nel corso della marcia di ritorno (°) non è previsto alcun ripensamento, dal momento che la KBD non è più invocata.

L'altra rozzezza, più sottile ma per certi versi più irrealistica ed inaccettabile deriva dal fatto che, quando si dà il comando 'R' può avvenire che si trovi in un istante corrispondente ad esempio ad un punto intermedio tra la stazione '0' e la '1'. Un'impostazione un po' più corretta della simulazione potrebbe essere la seguente (si tratta delle istruzioni che seguono il rilevamento della condizione C = 0):

```
          XOR A
          CP B            B = 0?
FINCORS   JP Z,ATTCOM     se SI halt, attendi comando
ANCOR5    CALL KDEL       aspetta ancora 5 msec. (°°)
          INC B           N.B. non DEC B!
          JR NZ,ANCOR5    continua attendere 5 msec.
          JR FINCORS      finché, con B = 0, fine corsa.
```

ATTCOM (v. oltre, codifica) è la label accanto alla CALL KBD iniziale, mentre l'INC B funziona così: se B, all'arrivo di 'R' era, mettiamo, per far un conto semplice, 250, dato che B nel corso del ritardo/polling *retrocedeva*, partendo inizialmente da 00 (come dire 256), vuol dire che si sono fatti, per esprimersi così, sei passi indietro. Occorrerà quindi per far passare altrettanto tempo di ritorno, contare *in avanti* (per tornare indietro! che bisticcio...): 251, 252,...255, cioè FFH, e, infine 00.

Perché, nonostante questo lasciamo le cose così? Soprattutto per pigrizia, e anche perché...tanto tutte queste finezze non si vedrebbero sullo schermo!

(°) - Si è simulata una velocità di rientro doppia del normale, senza alcun particolare motivo, tranne il fatto che tale RIT serviva anche nella parte di lampeggio stazione servita. Così facendo si soddisfa anche la massima del poeta barocco: "è del poeta il fin la maraviglia" (il frettoloso rientro rassomiglia a quello degli operai alla fine del turno).

(°°) - Trascurando gli 1,5 msec. di KBD che volendo si può anche inserire, col "polling" e tutto.

La codifica che segue non presenta praticamente alcuna particolarità che il lettore non sia a questo punto in grado di ... de-codificare da solo (si tratta di trucchi ormai noti ed arcinoti).

Programma P. 8-3

Indir.	Contenuto	Label	Cod. Assembly	Commenti
		INIZ	EQU 0BC0H	posizione d. "top row",
			ORG EF0H	(tanto per cambiare)
0E00	16 FF	CARPONT	LD D,FFH	valore "inesistente" in D
E02	21 CD 0B		LD HL,INIZ	punta all'inizio
E05	0E 30		LD C,'0'	stazione di partenza
E07	71		LD (HL),C	in C e sullo schermo
E08	1E 00	RESET	LD E,0	reset AV & IND
E0A	CD 69 00	ATTCOM	CALL 69H	chiama KBD, attendi tasto
E0D	30 FB		JR NC,ATTCOM	
E0F	FE 52		CP 'R'	è mica 'R'?
E11	28 F7		JR Z,ATTCOM	che c'entra 'R'? va ATTCOM!
E13	FE 4D		CP 'M'	magari è 'M'...
E15	28 03		JR Z,MARCIA	davvero? in MARCIA!
E17	57		LD D,A	allora è prenota o metti in D
E18	18 F0		JR ATTCOM	e torna ad att. comando
C1A	CB FB	MARCIA	SET 7,E	set flag AV
E1C	06 00		LD B,0	
E1E	7A		LD A,D	prenotazione in A
E1F	BE		CP (HL)	cfr. con posiz. attuale
E20	20 10		JR NZ,CONTIN	se diversa, CONTINua
E22	16 FF		LD D,FFH	valore ines. in D
E24	3E 0A	LAMP	LD A,10	A = 10
E26	36 20		LD (HL),' '	oscura la stazione
E28	CD 7E 0E		CALL RIT	per circa 0,30 sec. di RIT
E2B	71		LD (HL),C	rimetti valore in (HL)
E2C	CD 7E 0E		CALL RIT	tienilo per altri 0,65 sec.
E2F	3D		DEC A	ripeti per altre
E30	20 F4		JR NZ,LAMP + 2	dieci volte
E32	CD 35 00	CONTIN	CALL 35H	chiama KDEL
E35	CD 69 00		CALL 69H	chiama KBD
E38	30 0A		JR NC,DECB	se no caratt. va a DECB
E3A	FE 4D		CP 'M'	dici: 'M'?! ma ci siamo già
E3C	28 06		JR Z,DECB	va a DECB, se 'M'!...
E3E	FE 52		CP 'R'	comando 'R' (itorno)?
E40	CA 61 0E		JR Z,RITORN	se si va a RITORN
E43	57		LD D,A	è cifra: posa in D
E44	10 EC	DECB	DJNZ CONTIN	continua ciclo finché B = 0
E46	36 20		LD (HL),"	oscura posizione attuale
E48	CB 7B		BIT 7,E	com'è AV?

Indir.	Contenuto	Label	Cod. Assembly	Commenti
E4A	28 04		JR Z,GIU	se AV = 0 va GIU
E4C	23		INC HL	se AV = incrementa posiz. e il
E4D	0C		INC C	reg. C conta-stazioni
E4E	18 02		JR GIUGIU	poi va GIUGIU
E50	2B	GIU	DEC HL	con AV = 0 decr. pos. e
E51	0D		DEC C	reg. C
E52	71	GIUGIU	LD (HL),C	C sullo schermo
E53	79		LD A,C	compara
E54	FE 30		CP '0'	C con '0' se uguali, chiama
E56	CC 740E		CALL Z,INNANZ	rout. che: AV = 1; IND = 0
E59	FE 39		CP '9'	compara con '9' e segue ug. ALL
E5B	CC 79 OE		CALL Z,INDRE	rout. che fa AV = 0 e IND = 1
E5E	C3 1C 0E		JP MARCIA + 2	JP a: LD B,0 ecc. (ind. E1C)
E61	CD 79 0E	RITORN	CALL INDRE	inverti rotta
E64	3E 30		LD A,'0'	A = zero ASCII
E66	B9		CP C	è C = '0'?
E67	CA 08 0E		JP Z,RESET	se SI, va a resett. AV,IND
E6A	CD 7E 0E		CALL RIT	attendi 0,3 sec. circa
E6D	36 20		LD HL,' '	blank su questi schermi
E6F	0D		DEC C	
E70	2B		DEC HL	
E71	71		LD (HL),C	altra staz. in (HL)
E72	18 F0		JR RITORN + 3	va a cfr. C con '0' ecc.
E74	CB FB	INNANZ	SET 7,E	set AV
E76	CB B3		RES 6,E	reset IND
E78	C9		RET	
E79	CB BB	INDRE	RES 7,E	reset AV
E7B	CB 73		SET 6,E	set IND
E7D	C9		RET	
E7E	C5	RIT	PUSH BC	per salvare B
E7F	08		EX AF,AF'	salva Acc.
E80	06 40		LD B,64	
E82	CD 35 00	ALTGIR	CALL 35H	soliti 5 msec.
E85	10 FB		DJNZ ALTGIR	x 64 = 0,32 sec.
E87	C1		POP BC	recupera B
E88	08		EX AF,AF'	ed A
E89	C9		RET	fine RIT e programma

La codifica che precede richiedeva solo un po' si attenzione circa l'uso della routine del NASBUG detta KDEL. Come si può constatare tornando al cap. 3, *da essa si esce sempre con Accumulatore nullo*. Per questo motivo è stato necessario, nella routine RIT che impiega KDEL far precedere e seguire l'istruzione EX AF,AF' (altrimenti, nelle istruzioni dalla label LAMP e

seguenti il de-conteggio di A non funziona e anziché il lampeggio si ha buio fisso). Analogamente è indispensabile che, alla label CONTIN, la chiamata di KDEL preceda la CALL KBD: in caso contrario, in corrispondenza dell'indirizzo E38 si avrebbe *sempre* (cioè anche quando un tasto è battuto) A = 0 e Cy pure nullo!

La stessa situazione si è presentata nel programma P. 6-1 (semaforo), all'etichetta LAMP (si noterà che a rigore la Fig. 6-1 è invece errata, mentre la codifica, che è quella che conta, è giusta).

Nel congedarci dalla simulazione di congegni che vanno e vengono, proponiamo a chi ne avesse tempo e desiderio altri possibili esercizi:

- ascensore con piano "nobile" (che so, gli uffici delle alte sfere...) che, se prenotato, ha precedenza assoluta;
- carro-ponte a due sole stazioni, magari con comando "Halt" per la fermata d'emergenza in qualunque momento, con il viaggio rappresentato sullo schermo dallo spostamento a velocità opportuna di un carattere a piacere come l'asterisco o, più bellino ancora, una freccia volta a destra nella marcia avanti, a sinistra in quella indietro. I relativi codici sagittali sono rispettivamente 09 e 0D.

Con un po' di fantasia, le varianti su questi temi sono infinite.

Può essere interessante citarne una trovata empiricamente, quasi per caso: basta sostituire le istruzioni all'indirizzo E22 (ossia 16 FF) con due NOP (00 00, cioè) per ottenere un servizio *continuativo* della stazione prenotata, ossia *essa lampeggia tutte le volte che vi si passa*, sempreché nel frattempo non sia stata prenotata una diversa stazione. Il programma P. 8-3, come risulta abbastanza evidente, serve invece la stazione prenotata *una volta sola*.

A chi legge l'ardua sentenza su quale delle due soluzioni è più realistica, nonché il piacere di scoprire perché la semplice modifica suddetta comporti questo diverso...comportamento (cacofonia messa di proposito). Si fa solo notare che il caricamento del valore "inesistente FF in D, nell'istruzione all'indirizzo E00 resta necessaria anche nel caso di modifica con i due NOP all'indirizzo E22.

CAPITOLO 9

CONTROLLO DI UN ASCENSORE (QUASI REALISTICO) PROBLEMI DI INPUT/OUTPUT

Dopo tante simulazioni, che continuiamo a ritenere utilissime in sede di apprendimento, in quanto: a) si ragiona intorno alla logica essenziale di un problema, esercitando anche quelle doti di inventiva che abbiamo definito fin dall'inizio importanti per un programmatore; b) si impara l'uso delle istruzioni del microprocessore e, a volte, può anche trattarsi di impiegare trucchi non privi di sofisticazione ed eleganza; riteniamo però giunto il momento di tentare un approccio ai problemi in modo più aderente a quella che può essere la realtà impiantistica. Diciamo subito che essendo principalmente didattico e, prevalentemente, orientato al software il fine di questo libercolo, non ci si aspetti una trattazione tecnicamente molto approfondita specialmente nei dettagli particolari di questa o quella applicazione.

Comunque cercheremo di continuare il discorso iniziato al Capitolo 7 sui problemi dell'input/output e sulle interfacce tra il mondo esterno ed il microprocessore.

Ascensore (quasi) realistico (Programma P. 9-1)

Immaginiamo un impianto semplice, senza prenotazoni, utilizzante una sola unità Z80-PIO. Le connessioni si faranno come schematizzato nella figura 9-1, con le linee del porto A adibite alla chiamata e quelle del porto B connesse ad opportuni sensori posti in prossimità di ogni piano.

Circa le prime, dobbiamo far notare che a ciascuna di esse sono collegati due pulsanti di chiamata: uno entro la cabina ed uno sul pianerottolo del piano corrispondente. Tra di essi verrà quindi fatto l'OR logico (o mediante un vero e proprio *gate* integrato o semplicemente collegando i due pulsanti in parallelo, come tutti gli elettricisti sanno fare, anche ignorando l'esistenza di Mr. George Boole). In altri termini continuiamo, per semplicità, a non fare distinzione tra chiamate esterne ed interne il che, anche se probabilmente non è corretto, non ci interessa (°) anche perché, ora, dobbiamo pre-

---

(°) - Speriamo di non accendere in giro focose diatribe ascensoriche. In molti impianti del genere, comunque, la chiamata esterna prenota la cabina *se in marcia* e questa va al piano prenotato *quando vuota* (sulla base di questo criterio si può fare una simulazione sul NASCOM, con 5 tasti "interni" 5 "esterni" e, ovviamente, altrettanti piani.

cisarlo subito, non entriamo nel cuore di un progetto di controllo di ascensori ma vogliamo solo accennare a taluni problemi di I/O che in base a considerazioni elementari si può immaginare esistano su impianti del genere, più che altro assunti quindi come punto di riferimento.

Quanto ai sensori di prossimità si possono avere dai più semplici congegni del tipo fine-corsa elettromeccanico a più moderni dispositivi magnetici, a cellula fotoelettrica et similia.

Per poter fissare le idee, facciamo conto che l'attivazione di uno di tali pulsanti o sensori significhi valore "alto" al corrispondente ingresso della PIO. Così le cose sono più immediate e semplici. Supponiamo poi che il porto A sia indirizzato con 0 e B, connesso ai sensori, con 1.

È allora chiaro che, su entrambi i porti, sono possibili, almeno in condizioni normali (cioè nessun pulsante "incastrato" o sensore guasto) le seguenti informazioni:

- tutti i bit sono nulli = nessun pulsante azionato o sensore attivato;
- uno solo tra i bit è alto = un piano è chiamato o si è giunti ad uno di essi.

Siamo cioè in presenza di un *codice* particolare che nella *Teoria dell'Informazione* è solitamente denominato "one-out-of..."; nel nostro caso, al posto dei puntini andrebbe: "eight", mentre è classico il caso delle schede perforate in cui si usa un codice uno-di-dieci. Possiamo abbreviare con 1/8 (uno su 8).

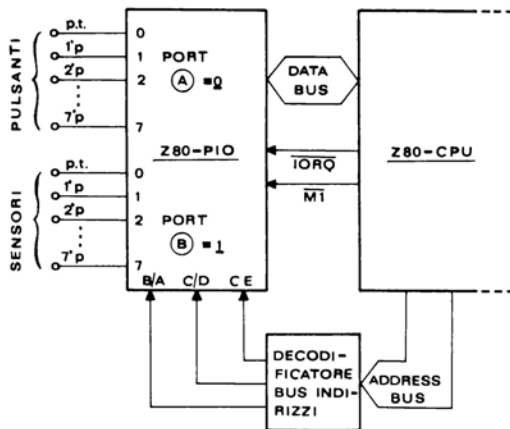



Figura 9-1 Possibile schema dell'input per 8 piani (o 7 + p.t.)

Anche in questo codice, comunque, sono possibili ed hanno senso le operazioni di comparazione: quando ad esempio si ha il dato in codice 1/8 corrispondente al 6° piano, in binario si ha (il bit 7 è naturalmente il più si-

gnificativo ed è connesso al piano omonimo)): 01000000, che in esadecimale è compatto in 40H ed è, per esempio, più grande del dato (binario) 00100000B (= 20H) e minore solo di 10000000B (= 80H), settimo piano). Si noti come si ha a che fare con tutte potenze del 2.

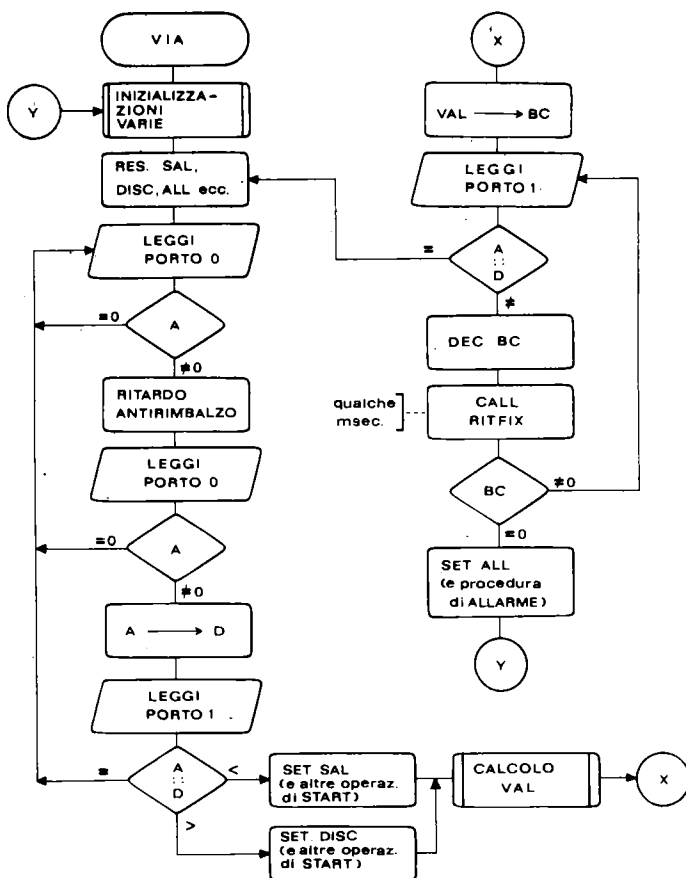



Figura 9-2 Flow chart di massima per ascensore 8 p. (imperfetto)

Pertanto il flow chart generico del nostro controllo di impianto ascensorico può tranquillamente recare il confronto tra il dato esistente nell'Accumulatore, dopo l'operazione di input ivi indicata con "LEGGI PORT 0" o l'altra "LEGGI PORT 1", e zero o qualunque altro valore (per esempio quello di una eventuale prenotazione).

Il flow di Fig. 9-2 che, lo ripetiamo, è molto semplificato e schematico,

da principio reca un sottoprogramma comprendente tutte quelle operazioni eseguite all'atto della messa in opera dell'impianto e/o nelle condizioni di alimentazione di tensione al nostro sistema di controllo a micro. Ci saranno anche istruzioni di inizializzazioni della PIO, secondo quanto visto al Cap. 7. Vengono poi resettati i bit denominati genericamente SAL e DISC nonché un terzo, ALL, la cui funzione vedremo tra poco.

La lettura del porto 0 è seguita dal confronto con zero, ossia tutti i bit nulli, il che significa assenza di prenotazione, nel qual caso l'istruzione IN A,(0) è eseguita di nuovo. Abbiamo così un semplice loop d'attesa di una chiamata. Quando questa avviene, sarà A = 0. Segue un ritardo software di 5 - 10 msec. in funzione *"antirimbalzo"* (inglese debouncing"): esso serve cioè ad attendere che i tremolii inevitabili nello schiacciamento di un pulsante (con la sua molla ecc.) nonché i fenomeni del transitorio elettrico si siano assestati affinché il dato sulla linea sia stabile e possa essere accettato con un buon grado di attendibilità. Qui si è anche rafforzata tale funzione mediante una successiva ri-lettura del medesimo porto, seguita da un confronto, nuovamente, con zero. Se A = 0, si torna alla prima istruzione di lettura-polling del porto. Ciò potrà presentare l'inconveniente che se il tasto è schiacciato troppo brevemente la chiamata è ignorata, ma una decina di millisecondi sono davvero pochi, comunque negli ascensori reali la gente è ben abituata a tenere pigiati i tasti, finché non compare la scritta "in arrivo" o "pronto".

Un mezzo ancora più energico di debouncing potrebbe essere il seguente:

```
LEGGI      IN A,(0)        leggi porto 0
           CP 0            cfr. A con zero
           JR Z,LEGGI      torna a LEGGI se A = 0
           LD B,A          dato letto in comodo B
           CALL MSEC       MSEC = ritardo di alcuni msec.
RILEGGI    IN A,(0)        fa un'altra lettura da (0)
           CP B            cfr. con valore prec.te
           JR NZ,LEGGI     accetta solo se concordano
           . . . . . . .   seguito programma.
```

In tutta franchezza, non pensiamo di avere l'autorità sufficiente per dire quali sono i pregi e i difetti dei due metodi. L'esperienza insegna che nelle applicazioni non eccessivamente veloci non creano problemi (e dovrebbero anche creare un certo effetto filtrante contro i disturbi di tipo casuale che si presentassero sulle linee).

Torniamo al flow di Fig. 9-2.

Quando A risulta diverso da zero e si supera l'esame rimbalzistico, il dato è messo nel registro-deposito D, poi si va a leggere dal porto 1.

Qui non si sono messe istruzioni di debouncing, il che appare ragionevole, dato che consideriamo ferma la cabina e, quindi, più che stabile l'indicazione dei sensori di piano.

Piuttosto, nelle ipotesi che stiamo facendo, sarebbe importante mettere dei controlli che impediscano la partenza (segnalando l'anomalia all'esterno) quando ci fosse un dato caratterizzato da più di un "1". Questa situazione creerebbe infatti un guaio serio con il flow chart di Fig. 9-2: poiché dal porto 1 è praticamente impossibile che arrivi un dato uguale, la cabina non potrebbe trovare in alcun piano un "ubi consistam"!

Proseguendo con (moderato) ottimismo, si va al confronto tra A e D, i cui possibili esiti sono del tutto analoghi a quelli già visti in sede di simulazione: con A = D si ignora la chiamata spiritosa, con A minore o, rispettivamente, maggiore di D si fa il set di AV o di DISC, rispettivamente, ponendo anche in opera, nei due casi, le opportune procedure connesse alla partenza in salita e in discesa.

Attenzione a non porre analogie del tutto meccaniche con gli esempi di simulazione visti: ora A significa quel che là abbiamo chiamato PATT mentre la chiamata è memorizzata nel deposito D!

Si noterà come finora, anche se molte sono le...mutanda che vanno mutate, la struttura essenziale di questo diagramma segue abbastanza da vicino quella dei flow visti negli esempi di simulazione al Cap. 8.

A questo punto, la parte di ritardo con il de-contatore BC che là ci serviva a fingere che la cabina stesse viaggiando, qui sembra del tutto inutile. Abbiamo ugualmente pensato di utilizzarla. Vediamo come. Si pone in BC un opportuno valore iniziale VAL, quindi si va a leggere dal porto 1, facendo seguire il confronto con D in modo che, quando si ha concordanza tra i due dati si vada al reset di SAL e DISC ad alle opportune precedure di arresto motore eccetera. Con A e D diversi si fa il decremento di BC, la chiamata di una certa RITFIX (parente del nostro vecchio KDEL) e, finché BC è diverso da zero si torna a rileggere il porto 1. Quando dovesse avvenire che BC = 0 si fa il set di un bit genericamente detto ALL al quale è associato l'inizio di acconce operazioni di allarme ecc. (il connettore Y indica ritorno a opportuno punto intermedio dell'inizio). Perché tutto questo? Orbene, possiamo pensare che VAL sia tale che, moltiplicato per RIT produca un tempo abbondantemente *maggiore* (ma non troppo) di quel che con funzionamento regolare (magari relativamente lento, ma entro i limiti previsti) avrebbe dovuto impiegare. A tale scopo però, prima di collegarsi al punto X, viene effettuato il calcolo del VAL che deve essere proporzionale al numero di piani da raggiungere.

Pur dovendo avvertire che non abbiamo alcuna pretesa di suggerire a nessuno di adottare questa soluzione di controllo della regolarità della marcia degli ascensori, al tempo stesso sottolineiamo: 1) che un meccanismo

consimile è comunque di largo impiego nelle applicazioni industriali (problemi già detti di "time out"); 2) è curioso come il lavoro di fantasia non sia poi così ozioso come potrebbe apparire agli spiriti pedanti (e limitati!).

Vediamo subito, visto che sicuramente abbiamo suscitato la curiosità di chi (ancora) ci segue, come fare tale calcolo. Chiaramente va fatta la differenza *in modulo* tra D e A e la cosa più semplice sarà, ad esempio, mettere il maggiore dei due nel registro B e il minore nel registro C. E poiché il confronto tra A e D è effettuato già, per stabilire se scendere o salire, la cosa migliore è di porre A in B e D in C nella catena in cui viene fatto il set di DISC e *viceversa* nell'altra in cui si accende SAL.

A questo punto però non conviene fare la differenza tra i due dati, perché sono in codice 1/8, ma occorrerà prima convertirli in binario normale.

L'algoritmo è semplice: si sfrutta l'operazione RRCA (Rotazione circolare destra) e si esce dalla routine se c'è Cy = 1, altrimenti si incrementa il registro del risultato, utilizziamo a tale scopo E fatto inizialmente uguale a zero (°). Una possibile codifica Assembly è la seguente:

```
                              ......
                              LD A,B           carica B in A (per RRCA)
                              CALL CONVER      routine di conv.ne
                              LD B,E           risult. conv. in B
                              LD A,C           carica in A C, ora
                              CALL CONVER      converti (risult. in E)
                              LD A,B           togli a B (conv.) il valore
                              SUB E            conv. di C, ora in E
                              LD H,0           trasferisci risultato
                              LD L,A           nella coppia HL (H = 0!)
                              ADD HL,HL        moltiplica per 2
                              ......           un certo N° di volte...
                              LD B,H           poi carica in
                              LD C,L           BC.
                              ......           resto del programma
routine        CONVER         LD E,0           azzera campo risultato
               ROTDEX         RRCA
                              RET C            se Cy = 1 rientra
                              INC E            altrim. incrementa ris. e
                              JR ROTDEX        torna a estrarre altro bit
```

(°) - Dato che stiamo facendo pezzi di programmi separatamente, senza preoccuparci dell'insieme, a titolo puramente indicativo, abbiamo supposto che ora il registro E non serva per i flag. In un programma complessivo si dovranno eventualmente inserire opportune istruzioni PUSH (poi POP) di salvataggio dati che servono più avanti nel programma.

Circa la conversione: un dato come 00...01 (cioè bit $A_0 = 1$, che corrisponde al piano terra) produce D = 0 (al primo giro è Cy = 1 dopo RRCA), come è giusto. Quanto alla istruzione ADD HL,HL che, come detto nel Cap. 2, genera il raddoppio del valore in HL, va ripetuta un numero opportuno di volte affinché al termine si abbia nella coppia e, poi in BC, il VAL opportuno (è un modo per evitare la moltiplicazione dato che qui una grande precisione non è richiesta).

Per completare il discorso, sia pure entro i limiti schematici dell'impostazione fatta, ci restano da vedere delle istruzioni per il controllo di dati di input con più bit "1" (così i ragazzini che schiacciano due pulsanti contemporaneamente sono sistemati, meglio ancora, e scherzi a parte, anche la routine CONVER su vista funziona con la massima sicurezza).

Si potrebbe seguire il procedimento consistente nel contare, facendo ruotare i bit, gli "uni" accettando solo dati che ne hanno uno soltanto. Ci piace si più la routine seguente:

```
          LD C,A          salva chiamata in comodo C
          LD A,1          00000001 in Acc.
COMPAR    CP C            cfr. con chiamata
          JR Z,LEGALK     se uguali, key legale, và avanti
          ADD A,A         genera altro codice legale
          JR NC,COMPAR    se Cy = 0 va a cfr. con C
          JR LEGGI        se no, rileggi porto 0
LEGALK    ......          (resto del programma)
```

Come appare evidente, l'algoritmo consiste nel generare in A, partendo da 00...1, poi con un raddoppio 00...10 e così via, tutti i codici "legali" confrontandoli con dato letto dal porto 0. Se uno di essi coincide con tale dato esso è accettato, altrimenti al termine, cioè quando al raddoppio di 10000000 si ottiene A = 0 con Cy = 1, si torna a leggere da tale porto. Grazioso, vero?

Hardware per l'emissione, il MEMORY MAPPING

Vediamo ora un (minimo) di hardware per l'emissione. A prima vista, dal diagramma a blocchi di Fig. 9-2 potrebbe apparire che i dati in emissione non ci siano, ne abbiamo comunque solo accennato brevemente. Anzi inizialmente qualcuno sarà trasalito sentendo parlare di una sola unità PIO per *tutto* il sistema. Non si tratta di una dimenticanza. Infatti, anche per ragioni di completezza della trattazione, anziché di un'altra unità di I/O (o del più banale attacco diretto dal Data Bus che naturalmente è *sempre* possibile previa *bufferizzazione* di esso per accrescerne il *fan out*, ovvero il numero di carichi TTL che il Bus può pilotare, notoriamente limitato a uno) vogliamo qui parlare della tecnica denominata comunemente *"Memory mapping"*.

Nella maggior parte dei problemi di controllo più semplici i ben 64 kilobyte di memoria indirizzabili sono ben lungi dall'essere impiegati: i programmi che abbiamo impostato in questo testucolo, anche con l'arricchimento di tutte le varianti possibili sono lunghi un duecento byte! Ma persino in tanti problemi industriali non si superano i 5 - 10 Kbyte).

Possiamo allora considerare un gruppo di bit di emissione come facenti parte di una cella di memoria, la cui realizzazione fisica è attuata con dispositivi, di solito, D-latch. Sono questi dei flip flop avanti un ingresso D (sta per *Data*, termine che gli anglosassoni usano per: dato) sul quale un bit di input 0 o 1 viene caricato sull'uscita Q (mentre, come in tutti i flip flop elettronici c'è anche un'uscita Q contenente il dato invertito) sotto l'azione di un altro ingresso detto "*Clock*" CK. Un'altra entrata è solitamente presente, molto spesso attivata dal livello basso, che forza sull'uscita Q il valore 0 ed è perciò detta entrata o comando di Reset o di "Clear".

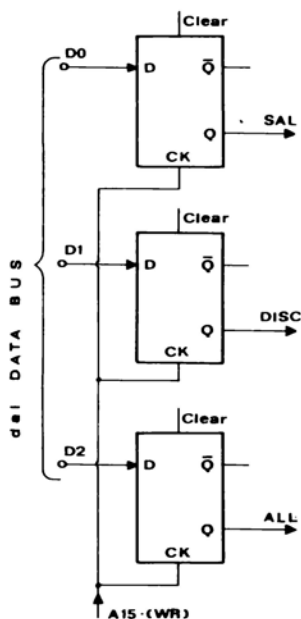



Figura 9-3 Uscite "memory mapped".

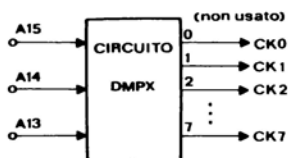



Figura 9-4 Multiplexer per indirizzamento di 8 periferiche (mem. mapp.).

Nel nostro semplice esempio abbiamo almeno tre linee di comando in uscita: SAL e DISC, che, oltre che funzioni di deviatoi di programma, hanno ora il senso preciso del comando l'uno dalla marcia avanti e l'altro della marcia indietro (e con SAL = DISC = 0 si comanda l'arresto); inoltre il bit

che abbiamo denominato ALL è anch'esso ipotizzabile come segnale di azionamento di qualche opportuno relé e/o dispositivo di segnalazione.

Operando allora i collegamenti di tre D-latch come nella figura 9-3, con CK = 1, ossia quando la linea dell'Address Bus A_{15} di peso maggiore è alta, verrà emesso il gruppo di tre bit $D_0D_1D_2$ sulle corrispondenti uscite dei latch denominate SAL, DISC e ALL. Ciò avverrà nel corso di una operazione di scrittura in memoria. Se quindi si riflette bene, abbiamo creato una piccola memoria di tipo né ROM né RAM, ma, per così dire, *di sola scrittura.*

È pure possibile, anche se ci risulta un po' meno usata, una tecnica diciamo duale con dei latch in input.

Per uno "strobe" dei dati, si potrebbe interporre un gate AND a due entrate: l'una la linea A_{15} e l'altra il segnale WR, che come è noto è attivo durante il ciclo di scrittura (tramite inverter, dato che dalla CPU proviene una linea WR). Meglio ancora, utilizzare integrati latch multipli già provvisti di un piedino del tipo "Chip Enable", come le RAM vere e proprie.

Vediamo come funziona l'indirizzamento in questa mini-memoria: la scrittura e, quindi, il trasferimento (dell'accumulatore o altro registro) sul Data Bus e, di qui, in uscita avviene tutte le volte che si indirizza una posizione di memoria avente il bit più alto uguale a 1. Ciò funziona in tutti i casi in cui la memoria totale impiegata, EPROM più RAM (per lo "scratchpad" e lo stack) non supera, come indirizzo massimo, il valore binario 100 .... 0 (quindici zeri), o, se si preferisce, eguaglia al più il valore (in esadecimale) 7FFF, vale a dire 32.767!. L'istruzione sarà:

LD (FFFFH),A o simili

Questa estrema semplicità è però possibile solo quando c'è da indirizzare un solo gruppo di otto flip flop al massimo (nel nostro esempietto addirittura i bit da D_3 e D_7 restano inutilizzati). Dovendo, in altri casi, indirizzare più dispositivi (di output o, come si è detto, anche di input) è necessario interporre un decodificatore indirizzi, solitamente un circuito LSI detto *Demultiplexer* (abbreviato: DMPX). Questo circuito converte da binario a 1/n, ossia attiva una sola linea di uscita alla volta.

Facciamo un esempio pratico: si supponga di poter sacrificare, per la memoria in senso stretto, i tre bit più pesanti $A_{15}A_{14}$ e A_{13}. Questo significa che il massimo indirizzo "reale" è 00011...1 (tredici "uno"), ossia 1FFFH = 8191, il che corrisponde, non si dimentichi l'indirizzo 00H, a 8192 celle, sufficienti per un progetto di media complessità.

Con i tre bit detti si possono allora selezionare sino a *sette* gruppi di D-latch, come illustrato nella Fig. 9-4.

Con $A_{15}A_{14}A_{13} = 000$ si ha la sola uscita $CK_0 = 1$ e tutte le altre nulle; con $A_{15}A_{14}A_{13} = 001$ solo $CK_1 = 1$, con gli stessi ingressi che valgono 010 è solamente CK_2 attivato e così via. Si noti bene che l'uscita che solo per omogeneità con le altre abbiamo denominato CK_0 (nel chip sarà indicata con Q_0 o simili) *non deve* essere usata perché il valore delle entrate del DMPX nulle si ha in *tutti* (!!) i casi di indirizzi di memoria veri e propri.

Tornando un momento alla figura 9-3, c'è da aggiungere che il comando d reset "Clear" potrebbe essere impiegato per un azzeramento manuale dell'esterno, il che solitamente va fatto con la massima cautela (spesso in fase solo di messa a punto o revisione da parte degli esperti) per una serie di motivi, che si possono intuire, e sui quali comunque non intendiamo aprire discussioni bizantine.

Tornando invece al nostro programmino, volendo esemplificarne l'uso del memory mapping per il set, ad esempio di SAL o di DISC, esso potrebbe essere congegnato come segue, nel punto in cui si fa la prima lettura del porto 0:

```
        IN A,(1)           leggi sensori
        CP D               compara con chiamata
        JR Z,LEGGI         chiamata fasulla
        LD HL,FFFFH        prepara puntatore: chiamata corretta
        CALL NC,DISCES     esegui routine DISCES se D < A
        CALL C,SALITA      esegui routine SALITA se D > A (p. att.)
        ........           (resto del programma)
```

le due routine saranno del tipo:

```
DISCES  LD (HL),20H        set bit 0 (SAL), reset bit 1 (DISC)
        ......             altre istruzioni
        RET
SALITA  LD (HL),01H        set bit 1, reset bit 0 (°)
        ......             altre istruzioni
        RET
```

Va fatta a questo punto un'osservazione importante (già preannunciata in precedenza): ora che siamo a conoscenza della tecnica del memory mapping possiamo dire che le nostre simulazioni non erano poi tanto campate in aria. Anche in progetti reali l'emissione avviene indirizzando in aree di me-

---

(°) - Per la verità questa allettante tecnica di accensione e spegnimento, in un sol colpo, di due flag, funziona solo se non c'è... un *terzo* incomodo, nel nostro caso il bit ALL che va lasciato immutato. L'abbiamo qui usata per farne una citazione. Tutto sommato le istruzioni di SET e RES sono preferibili anche quando ci costano qualche byte in più (*andavano usate anche qui*).

moria, così come abbiamo fatto noi, utilizzando aree della video RAM.

Questo anzi potrebbe suggerire tecniche "serie" di studio e debug anche di progetti reali, almeno in fase preliminare.

A questo punto ci si può chiedere, data la semplicità della tecnica del memory mapping, quale vantaggio presenta su di essa l'uso di una PIO.

Intanto c'è da notare che, dal momento che anche la prima tecnica richiede pur sempre un po' di hardware, tanto vale spendere un po' di più (e questo in un settore in cui i prezzi vanno crollando) e avere un congegno più potente che, un domani che il nostro progetto si complica (es. di richiede l'allacciamento di qualcosa che ha le caratteristiche di una periferica che per operare ha bisogno dell'"handshake") ci consente di far fronte solo modificando il software e qualche collegamento, lasciando immutato l'hardware della macchina. Comunque il vantaggio principale a nostro parere risiede nella possibilità che si ha con la PIO non solo di fare l'emissione su di un porto di uscita, *ma anche di rileggere in esso* (cioè sul registro di output dell'unità) il dato precedentemente emesso, quando si vuole (cosa in teoria possibile anche con dei D-latch, ma complicando ancora l'hardware).

Generazione di impulsi in uscita

Immaginando ora che in un sistema di controllo sia richiesto in uscita un *impulso*, cioè un segnale digitale che è attivo per una durata limitata, come può essere richiesto per pilotare un classico flip flop tipo R-S o, più ordinariamente, un comando di teleruttore. Questo, come è noto, funziona in modo del tutto analogo al primo: per l'azionamento o "marcia" il comando ha una durata limitata, tanto appena il relé si eccita subentra un contatto detto *autoblocco* che mantiene il relé autoeccitato (finché un comando impulsivo analogo, quello di "arresto" non giunge a diseccitarlo).

Comunque sia motivata questa esigenza, tecnicamente denominata anche *simulazione di un "one shot"*, il software è sempre schiavo del committente. La soluzione è facilmente intuibile. Per impulsi brevissimi:

a) direttamente sul Data Bus, senza PIO o simili:

```
        LD A,80H        (impulso, per es. sul bit 7)
        OUT (2),A       (dura circa quanto l'istruz. OUT)
```

b) impulso brevissimo, tramite unità Z80-PIO:

```
        LD A,80H
        OUT (3),A       set bit 7, porto 3
        XOR A
                        A = 0
        OUT (3),A       reset bit
```

i tempi sono quelli, esprimibile in microsecondi, dei dispositivi elettronici, quindi in grado di pilotarne uno del genere; con dispositivi elettromeccanici

è invece necessario interporre un tempo adeguato:

```
LD A,10H          set bit 4 (tanto per cambià)
OUT (4),A
CALL PAUSA        opportuno ritardo (vari msec.)
LD A,0
OUT (4),A
```

N.B. - Qui sopra, per brevità, non ci si è preoccupati di "mascherare" i bit diversi da quello attivato.

Immaginando siano necessari comandi impulsivi, nel caso dei nostri SAL e DISC, si comprenderà che non bastano più due bit: un teleruttore infatti necessita, come già detto, di *due* comandi (arresto oltre che marcia). Se pensiamo ad un terzo teleruttore per l'ALLarme, sul Data Bus restano solamente due bit (sempreché si voglia essere così tirchi da impiegare un solo circuito PIO e una sola cella di memoria-uscita). Immaginandoli al servizio di due lucine (a forma di freccia in su e in giù) per segnalare il senso di marcia, stavolta possono permettersi di restare "on" o "off" per tutto il tempo necessario. Sfruttando allora quella proprietà già detta della PIO di mantenere il dato e di consentirne la rilettura, i due bit possono *essere impiegati anche come deviatoi* di programma ed interrogati quando si vuole. Ad esempio:

```
IN A,(1)
BIT 0,A           eccetera.
```

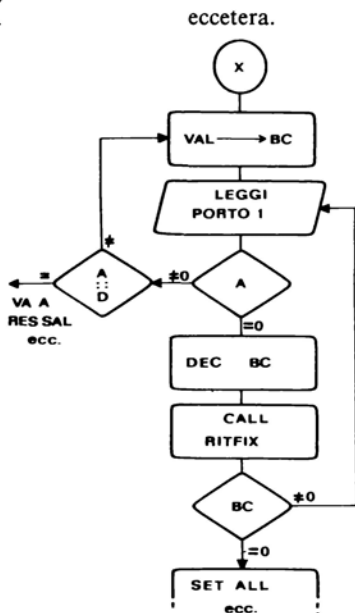



Figura 9-5 Variante più funzionale del flow di Fig. 9-2.

Semplificazione del flow chart di Fig. 9-2

L'impostazione del "time out" rappresentata in Fig. 9-2 con la tecnica del lasciar trascorrere un tempo proporzionale al numero di piani da raggiungere è stata escogitata, lo riveliamo ora, per consentire un discorso di calcoli e conversioni che ci è sembrato comunque interessante.

In realtà si poteva fare a meno di questa complicazione! La soluzione più semplice e naturale, che oltretutto ricalca anche meglio i programmi di simulazione visti al capitolo precedente è illustrata nella figura 9-5.

Appena fatta la lettura dal porto 1, si va innanzitutto a vedere se A è diverso da zero, il che implica *raggiungimento di un qualche piano* e se la risposta ad una tal domanda è positiva si confronta il nuovo piano con la chiamata: se c'è coincidenza, A = D, si va come prima all'arresto, altrimenti si torna ad inizializzare il doppio registro BC con un VAL che, stavolta, è semplicemente una *costante* relativa ad un piano solamente. Con A = 0 BC è decrementato e così via, giungendo al set di ALL e al resto quando BC = 0 che qui ha il significato di "cabina in ritardo rispetto al piano *precedente*". È chiaro che questa impostazione non è soltanto più semplice, ma ha pure il vantaggio di un controllo più frequente.

Nello scusarci con il lettore se, involontariamente gli abbiamo creato un po' di confusione (ma probabilmente, egli, accorto com'è, si era già avveduto della magagna) gli proponiamo un'altro possibile controllo temporale, da attuarsi all'arrivo ad un piano: quello di BC con un termine anch'esso opportuno (°), denotante invece viaggio *troppo veloce*... Qui entriamo nel complesso problema delle sicurezze e della spinosa questione di quanto, a tale riguardo, debba essere affidato all'hardware esterno e quanto al software. Per esempio: se i sensori di piano sono guasti, come può un programma (diverso naturalmente dal nostro) accorgersene, evitando che la cabina sbatta sul soffitto o in fondo al pozzo? Mediante controlli, diciamo, di "time in", la cosa è in linea di massima possibile, ma forse la sicurezza si accresce meglio con sistemi tradizionali: sensori di grande affidamento, magari in numero doppio, freni di emergenza automatici ecc.

Problemi di input con un numero maggiore di piani

Supponiamo di volere un massimo di 12 piani. Generalizzando meccanicamente quanto precedentemente visto, parrebbero ora necessari quattro porti, due per la chiamata e due per i sensori di piano. Ci sono però varie soluzioni più semplici, ricorrendo ad un po' di hardware esterno e mantenendo l'uso di una sola PIO.

(°) - Questo termine, dato che BC conta indietro, è un valore *troppo alto* che denota quindi un tempo trascorso troppo basso.

Una prima soluzione (Fig. 9-6) che ha il vantaggio di funzionare fino a 15 piani impiega un circuito, che, per così dire, concentra 16 linee in 4. Con i circuiti integrati LSI ora disponibili un tale dispositivo si implementa mediante due "*priority encoder*" del tipo 8 linee in 3 linee SN74148, che in sostanza opera la trasformazione da codice 1/8 a binario (es. se è attiva la linea 5, in uscita si ha il numero binario 101).

Qualora il numero massimo di linee di input fosse 10 basterebbe un solo integrato, il tipo SN74147 da 10 a 14 linee BCD.

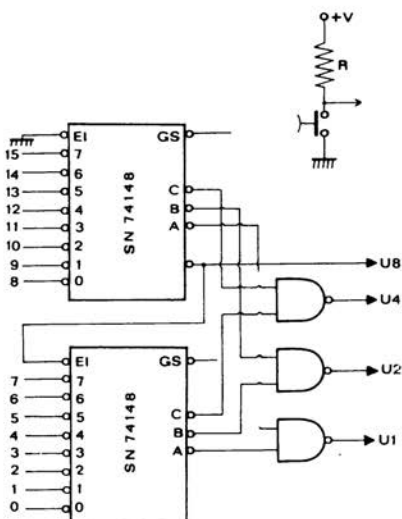



Figura 9-6 Priority Encoder per il concentramento da 16 a 4 linee.

Con la disposizione di Fig. 9-6, cioè con due 74148 e tre porte NAND, si ottiene un dispositivo che trasforma il codice 1/16 in binario. Per una spiegazione più dettagliata rimandiamo a qualche catalogo di IC (es. il TTL Data Book della Texas Instruments). Qui ci limitiamo ad osservare che, quando nessuna delle linee del priority encoder superiore è attivata, la sua uscita EO è bassa e come tale abilita l'altro, sulla linea EI. Si avrà U_8 (cioè il bit di peso 8, MSB) uguale a 0 ed i bit di peso 4,2 e 1 (U_4, U_2 e U_1) pari al numero della linea azionata tra quelle inferiori. Infatti, sempre nell'ipotesi di azionamento di un tasto inferiore o uguale a 7, le uscite A, B e C del 74147 di sopra risultano tutte alte ed i tre NAND fanno da invertitori.

Qui va richiamato il fatto che l'SN74148 dà in uscita valori invertiti (onde, se ad es. è azionato 3, l'uscita è ABC = 100, anziché 001); il circuito combinato che stiamo descrivendo dà perciò uscite positive (cioè premendo 3 si ha $U_8U_4U_2U_1 = 0011$ in uscita). Quando invece si attiva una linea superiore, per esempio 11, la EO, cioè U_8, va alta e inibisce l'IC inferiore, onde le uscite ABC di questo vanno ora tutte alte ed i tre NAND operano ora da inverter per le uscite dell'IC superiore: nell'esempio che abbiamo detto: ABC = 100 e quindi $U_8U_4U_2U_1 = 1011$.

Poiché le entrate sono attivate con valori bassi di tensione, i pulsanti saranno connessi come indicato in alto a destra della stessa figura.

Tornando al nostro ascensore, occorreranno naturalmente 4 IC SN74148 (più 6 NAND) due per i pulsanti di chiamata e altrettanti per i sensori di piano.

Osserviamo anche che la disposizione di Fig. 9-6 dà uscite nulle sia premendo il tasto 0 sia con nessun tasto premuto. Per evitare confusioni, allora, la cosa più semplice è di limitare il numero di piani a 15 (con l'eventuale p.t. codificato con 0001).

Per l'esattezza, l'SN74148 distingue i due casi con l'uscita GS che è bassa se un qualsiasi tasto è premuto (e, per generalizzare questa funzione al caso di 16 linee, basta far confluire le due linee GS in un quarto NAND (v. il citato catalogo TTL Data Book), ma tutto sommato la soluzione detta ci appare più semplice: ancorando al + 5V l'entrata 0 dell'SN74147 di sotto, l'uscita nulla indica "nessuna chiamata", direttamente.

Già che ne stiamo parlando, va sottolineata l'importanza di questo integrato in applicazioni a microprocessore (per fissare la priorità degli interrupt, ad es., anche se i circuiti standard della famiglia Z80 come PIO, SIO ecc. non ne hanno bisogno, operando in "daisy chain"). Il termine "priority" deriva dal fatto che esso, con più linee di input azionate simultaneamente "dà retta" solo a quella più alta (nell'applicazione che qui discutiamo questo può essere un vantaggio).

Suggeriamo poi una soluzione un po' artigianale ma economica che può funzionare, con una sola PIO, sino ad un massimo di 12 piani. Essa è illustrata nella Fig. 9-7. Si usano 3 *selettori di canali a 4 bit* tipo SN74157.

Questo integrato nei cataloghi è presentato come *quadruplo selettore da 2 linee in una:* è chiaro che, considerando a 4 a 4 le linee di input e di output e tenendo conto che il comando di selezione S è *comune*, lo si può anche chiamare, forse più utilmente, come abbiamo fatto noi.

Le entrate di ciascuno sono connesse, rispettivamente, alle 4 + 4 + 4 = 12 linee di chiamata e alle altrettante dei sensori di piano, così come si può vedere dalla detta figura, in modo che quando S ha un valore viene trasferito

sulle uscite il gruppo dei bit di chiamata, mentre, con S opposto, sono i bit dei sensori ad essere inviati alla PIO.

Le uscite dei primi due selettori sono collegate al porto A della PIO mentre quelle del terzo selettore vanno ai bit, per esempio, più pesanti del porto B. L'unità Z80-PIO deve qui operare in modo 3 (di Input/Output) col bit 3 del porto B adibito a comando di selezione S. Rimangono così liberi i tre bit 2, 1 e 0 che possiamo pensare di utilizzare in funzione dei già visti valori SAL, DISC e ALL (beninteso come segnali stabili e non impulsivi, per quanto già detto).

Vediamo, nei punti essenziali, come possono procedere le operazioni di input e di output.

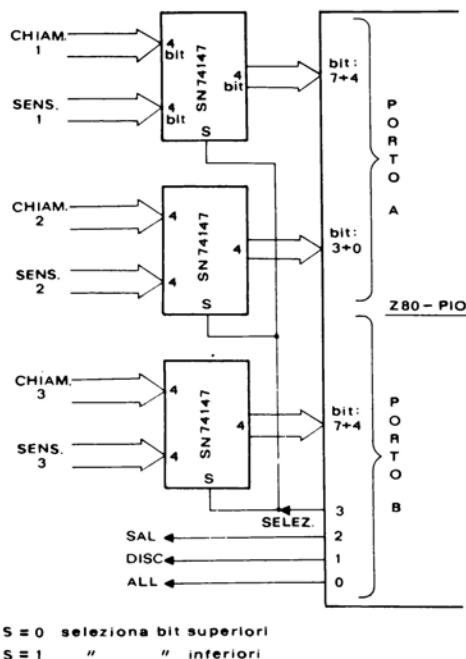



Figura 9-7 Uso di 3 selettori canali 4 bit per un massimo di 12 piani.

L'input e l'output nell'esempio di fig. 9-7

Per leggere una chiamata, supponendo che occorra il valore S = 0, si avranno anzitutto istruzioni per agire sui selettori tramite questo bit:

```
IN A,(1)        leggi porto B
RES 3,A         resetta ad
OUT A,(1)       emetti S = 0
```

Dello stesso tipo saranno le istruzioni per la selezione del gruppo sensori (con SET, anziché RES) e per l'attivazione o la disattivazione di SAL, DISC, ALL, ossia, dopo aver letto il porto B conviene agire sul solo bit interessato, in modo che la successiva operazione di emissione sullo stesso porto lasci immutato il valore degli altri bit.

A questo punto seguiranno le istruzioni per leggere i bit di chiamata (o dei sensori) ora disponibili sulle uscite dei tre selettori. Poiché si superano gli otto bit, conviene compattare i 12 bit in un solo registro: si tratta in sostanza di una conversione software da codice 1/12 a binario, il cui algoritmo è nella sostanza simile a quello già visto per il codice 1/8, anche se il fatto di avere ora in gioco due porti (per la nostra inguaribile mania delle varianti) ci suggeriscono di illustrarlo con un flow chart, disegnato nella figura 9-8.

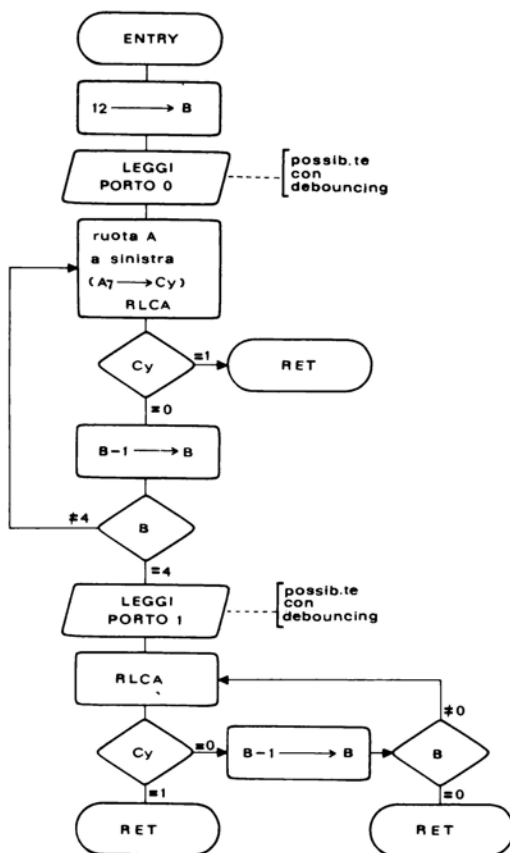



Figura 9-8 Flow chart della transcodifica piani (es. di Fig. 9-7).

Si pensa di utilizzare il registro B e di inizializzarlo con 12, in quanto la scansione successiva dei bit, con l'istruzione RLCA (cioè rotazione *sinistra*), è fatta partendo dal più alto.

Dopo IN A,(0), segue la detta RLCA poi se Cy = 1, ciò significherà, se già alla prima rotazione questo accade, che si tratta del 12° piano, pertanto si esce dalla subroutine, altrimenti si decrementa B ripetendo il loop fino a che non si ha B = 4. In quanto precede infatti era implicito che i collegamenti fossero fatti in modo che i bit dei piani 12, 11, fino a 5 corrispondano rispettivamente ai bit 7, 6, ... 0 del porto A, supposto indirizzato con (0).

A questo punto, con B = 4 bisogna passare al porto 1, cioè B. Ripetendo lo stesso meccanismo di prima è chiaro che le possibilità finali sono due:

- B è diverso da zero ed il suo valore corrisponde al piano chiamato (o raggiunto);
- B è nullo, il che significa che tutte le linee esterne sono disattive.

Per il modo come abbiamo impostato le cose, il piano terra, ove esistente, verrebbe codificato con 1. Si poteva anche partire con valore iniziale di B = 11, col che la condizione di "nessuna chiamata" sarebbe codificata con FF, anziché con 0 (che corrisponde a piano terreno). Naturalmente i due confronti di B presenti nel flow vanno fatti, rispettivamente con 3 anziché con 4 e con FFH in luogo di zero.

La codifica del tutto è scontata:

```
CONV12    LD B,12
          IN A,(0)
RUOT1     RLCA
          RET C            rientra se Cy = 1
          DEC B
          PUSH AF          (V. nota sotto): salva Acc.
          LD A,4
          CP B
          POP AF           recupera Acc. (chiamata!)
          JR NZ,RUOT1
          IN A,(1)
RUOT2     RLCA
          RET C
          DJNZ RUOT2
          RET              nessuna chiamata
```

Nota: PUSH AF è preferibile a EX AF,AF' che potrebbe mettere in Cy un valore...inopportuno.

È possibile forse trovare un modo più compatto di conversione. Comunque in genere, nei problemi di questo tipo, le operazioni di input/output si complicano un poco (per motivi di conversioni, debouncing anche più sofisticati di quelli che abbiamo citato ecc.). Ci vorrà pazienza e utilizzare su-

broutine abbastanza standard nell'ambito di un progetto o, meglio, di una serie di programmi di macchina simili. Così facendo, nel MAIN basterà citarne il nome (INPUT, PORTINP o simili).

Pulsantiere e tastiere

Le soluzioni dianzi proposte possono essere interessanti ai fini di una discussione. In genere però, quando si ha a che fare con molti pulsanti, come ad esempio nella tastiera di un sistema di sviluppo, la soluzione ottimale è rappresentata dallo schema *a matrice* del tipo illustrato nella figura 9-9, relativa al caso di 16 tasti. La meccanica è tale che, quando un tasto è premuto, esso mette in contatto un filo di linea con uno di colonna: ad esempio nella Fig. 9-9 premendo il tasto 7 si collega la linea c con la colonna B. Se ora un dispositivo demultiplexer che attiva con il livello alto (di solito + 5V) una ed una sola linea, nota ad ogni istante (meglio, ad ogni colpo del clock che pilota il circuito), uno dei quattro tasti della linea sarà in grado di trasmettere questo livello alto ad una ed una sola, ben precisa, colonna. Ad esempio, se è eccitata la linea c ed è premuto il tasto 6, sarà messo al livello di tensione + 5V il filo di colonna C: conoscendo quale riga è attivata in quel momento, il tasto messo in funzione in quel momento sarà allora individuato sulla base della colonna che è anch'essa a livello alto.

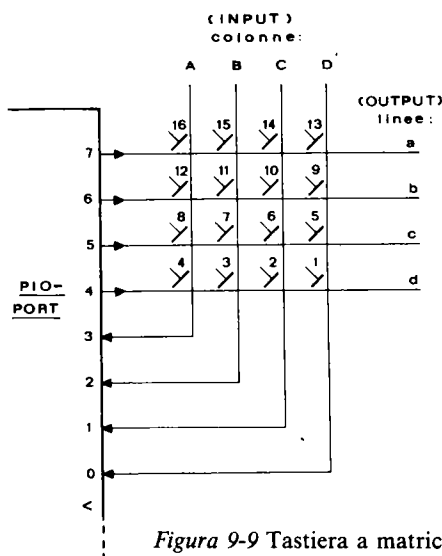



Figura 9-9 Tastiera a matrice a 16 chiavi con polling software.

Questo meccanismo può essere implementato interamente via hardware oppure parte hardware e parte software. Noi qui descriviamo la tecnica inte-

ramente software, che comporta, in cambio di maggiori complicazioni del programma, costi minori (questo almeno al tempo presente in cui ormai i costi delle memorie sono fortemente diminuiti, al punto che entro certi limiti - soprattutto i tempi di programmazione - la lunghezza di un programma non è più, nei medi progetti, un problema).

Come si vede dalla già citata figura, non si richiede nessunissimo gate (°).

Un solo porto della PIO è richiesto, di cui 4 bit sono impiegati come output ed attivati, uno alla volta, ciclicamente, mentre gli altri quattro sono oggetto di *polling*, ad uno ad uno, durante il tempo in cui una colonna è mantenuta a "1".

Flow chart

La soluzione che qui si propone sotto forma di una subroutine POLKEY (cioè POLling di una KEYboard), oltre che dell'Accumulatore, per le operazioni IN e OUT, ha bisogno dei registri B, C, D ed E. Il diagramma a blocchi è in Fig. 9-10.

Scopo della routine è di creare nel registro E un numero esadecimale da 1 a 16, a seconda del tasto premuto. Se nessuna chiave è azionata, si avrà E = 0.

Il primo blocco inizializza E al valore 16, B a 4 ed A a zero, mentre il flag Cy è messo in "on". Inizia poi un loop più ampio (etichettato con la label CICLON nella codifica, v. oltre), avente lo scopo di porre successivamente, nelle linee di output 7, 6, 5, 4 il livello alto. Ciò viene effettuato mediante l'istruzione RRA che opera la rotazione dell'Accumulatore *insieme* al Carry. Dato che abbiamo posto Cy = 1, è evidente che la ripetua esecuzione di RRA pone successivamente il livello alto solamente in un bit a partire da 7 in giù.

Lo stesso risultato si poteva anche conseguire con; LD A,01 seguito da RRCA (rotazione circolare destra. Si è solo voluto illustrare l'uso di un altro codice operativo (il numero dei byte è lo stesso nei due casi).

Con l'istruzione OUT (0),A questo livello alto è emesso sulle linee 7, poi 6, 5 e 4, come si è già detto. Dentro CICLON, che termina quando B, decrementato in fondo al sottoprogramma, è uguale a zero, è inserito il loop più interno CICLIN per la successiva lettura dei bit 3, 2, 1 e 0: prima si legge il

(°) - In realtà occorreranno dei buffer sui bit di uscita, non invertenti nelle ipotesi che stiamo facendo. Se invece si usano degli inverter, sarà necessario far "ciclare" dei livelli bassi.

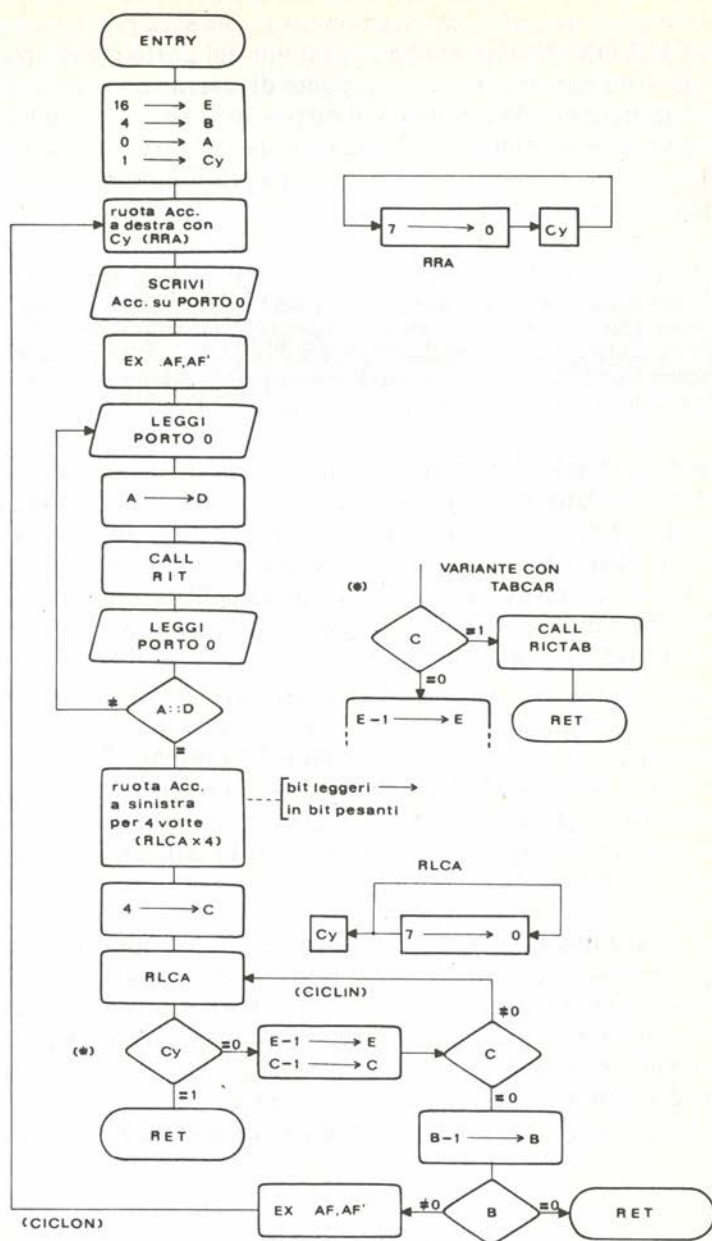



Figura 9-10 Flow chart del polling tastiera 16 chiavi.

porto 0, poi si fa questa scansione. Ancor prima, si salva, con l'istruzione EX AF,AF', il contenuto dell'Accumulatore che sarà poi necessario alla ripresa di CICLON. Notare anche che l'input del porto 0 è effettuato con la tecnica già vista antirimbalzo. Si è pensato di inserire un ritardo di solamente 2 msec. immaginando l'impiego di dispositivi ormai in commercio dotati di tasti con modesto effetto rimbalzo. Così da un lato la semplice tecnica qui impiegata dovrebbe essere sufficiente dall'altro il tempo massimo di scansione viene ridotto a 8 msec. (°)

Ripetiamo comunque che il metodo di debouncing con due letture intervallate da ritardo (per l'assestamento dei rimbalzi) e ripetizione in caso di discordanza tra i due successivi valori letti non è che una tra le più semplici del genere. Per quelle più complicate rimandiamo ai testi più approfonditi, bastandoci qui di aver introdotto il problema.

Tornando al nostro flow, siamo rimasti al punto in cui nell'Accumulatore è presente un dato i cui 4 bit più leggeri possono contenere l'indicazione di una chiave premuta, appartenente alla linea prima attivata. Si ricorda a questo proposito che, con la PIO operante in modo 3, come qui era evidentemente implicito, l'istruzione di input carica nell'Accumulatore anche i bit di emissione (mentre, come è giusto, l'istruzione di output *non* modifica certo i bit di input sui registri della PIO).

Occorre ora estrarre ad uno ad uno, nel carry, i bit 3, 2, 0 e 1, in quest'ordine. L'istruzione più adatta è la RLCA (ruota a sinistra circolarmente, senza comprendere il Cy entro la rotazione). Preliminarmente è più comodo (anche e soprattutto per il meccanismo di ottenimento in E del corretto valore esadecimale) spostare i detti bit leggeri al posto dei bit più pesanti 7, 6, 5 e 4. Ciò si farà con la ripetizione per quattro volte dell'istruzione RLCA (andava anche bene la ADD A,A pure ripetute 4 volte, ma così la cosa è forse più evidente).

Dopo aver fatto C = 4 segue un quinto RLCA, inserito entro in loop "piccolo" detto perciò...CICLIN, per la successiva estrazione in Cy di 7, 6, 5 e 4 che coincidono con i vecchi 3, 2, 1, 0. Con Cy = 1 si esce dalla routine con il valore inizialmente impostato E = 16, se questo esito si è verificato con la scansione della prima linea e prima colonna (bit 3, ora in 7, uguale a 1). Se invece ciò è avvenuto, sempre alla prima linea ma alla seconda, terza ecc. colonna si dovrà uscire con E = 15, 14 ecc., successivamente. A tale fine, subi-

---

(°) - N.B. Non, come potrebbe apparire di primo acchito, 32 msec. Infatti il tempo di debouncing è lasciato passare solo dopo ogni attivazione di colonna, come risulta evidente da un esame del flow.

to dopo il test del Carry, il registro E è decrementato (si osservi di nuovo la figura 9-9, in cui la disposizione dei numeri accanto ai tasti, che a prima vista poteva apparire un po' bizzarra, è invece stata fatta in funzione di questo algoritmo di deconteggio da 16 in giù).

Con il codice DEC C si passa, con C diverso da zero, ad una nuova colonna, ripetendo la RLCA (cioé esame del bit 6, ex bit 2 ecc.), fino a che, se non si trova Cy = 1, non si giunge ad avere C = 0.

A questo punto si entra nella parte terminale di CICLON, col decremento di B, per passare alla linea successiva. Con B = 0 la... danza (POLKEY!) termina con E = 0 e Cy = 0.

Questo fatto, cioé che Cy = 0 se nessun tasto è premuto ed è invece uguale a 1 quando ne è stato attivato qualcuno, può essere un mezzo utilizzabile nel programma chiamante la POLKEY per mettere in opera azioni opportune (es. ri-chiamare la POLKEY medesima). Tutto ciò in modo del tutto identico a come si è già utilizzata la routine KBD nei capitoli precedenti.

Codifica Assembly

Seguendo direttamente il flow chart che abbiamo illustrato, essa non presenta difficoltà al lettore ormai esperto:

```
POLKEY      LD E,16
            LD B,4              n° cicli ampi = 4
            XOR A
            SCF                 set Cy
CICLON      RRA                 attiva una
            OUT (0),A           linea di output
            EX AF,AF'           salva Acc. e Cy
LEGGI       IN A,(0)            lettura
            LD D,A              con
            CALL RIT            debouncer (2 msec.)
            IN A,(0)
            CP D
            JR NZ,LEGGI
            RLCA                sposta bit 3, 2, 1, 0
            RLCA                nei
            RLCA                bit
            RLCA                7, 6 ,5 ,4
            LD C,4
CICLIN      RLCA
            RET C               esci se Cy = 1
            DEC E               altrim. prova se
            DEC C               è un altro
            JR NZ,CICLIN        testo, stessa colonna
            DEC B               oppure tenta con altra colonna:
            RET Z               esci se fine colonna
            EX AF,AF'
            JR CICLON           torna a CICLON se ancora
                                vi sono colonne.
```

L'istruzione EX AF,AF', volendo, si poteva anche mettere prima di DEC B e fare (dopo JR NZ,CICLIN):

```
EX AF,AF'
DJNZ CICLON
RET
```

e non vi sono pericoli di sorta, dal momento che Cy, dopo la prima RRA (label CICLON) si trova e resta nullo).

Una variante più interessante nasce piuttosto dall'osservazione che, a ben riflettere, il registro B è superfluo, dal momento che quando B = 0 è pure E = 0. Si può allora eliminare E ed utilizzare C in sua vece e B al posto del registro C. Ossia, per non confondersi: C serve al numero che rappresenta in binario il tasto premuto, B conta i cicli "piccini".

La routine modificata si presenterà allora come segue:

```
POLKEY   LD C,16
         LD A,01          variante, al posto di XOR A,
CICLON   RRCA             SCF poi RRA
         OUT (0),A
         EX AF,AF'
LEGGI    IN A,(0)
         LD D,A
         CALL RIT
         IN A,(0)
         JR NZ,LEGGI
         RLCA
         RLCA
         RLCA
         RLCA
         LD B,4           inizializza cont. ciclini
CICLIN   RLCA
         RET C
         DEC C
         DJNZ, CICLIN     altro CICLIN se B ≠ 0
         LD A,C
         OR A             attiva i flag!
         RET Z            se C = 0, finer scansione
         EX AF,AF'
         JR CICLON
```

Come si vede, la routine risulta un poco più compatta, anche se la precedente è più lineare e chiara.

Dato che siamo in argomento, un altro piccolo, ma stavolta elegante perfezionamento si può ottenere interrogando il bit 4 prima di riprendere alla label CICLON: se esso è attivo, ciò significa che tutti i bit dell'ultima colonna sono stati saggiati (nell'ultimo CICLIN), pertanto si dovrà rientrare

dalla POLKEY con C = 0. Si avrà:

```
        ........
        DJNZ CICLIN
        EX AF,AF'           ripristina attivatore linee
        BIT 4,A             è stata attiv. linea 4?
        JR Z,CICLON         se NO, torna a CICLON
        RET                 altrimenti rientra.
```

Nel caso di una tastiera a 32 caratteri, relativa ad un terminale, sistema di sviluppo tipo il nostro e simili, si può fare una matrice di 8 linee e 4 colonne, impiegando un porto e mezzo, il primo, ad esempio, come output verso le linee della matrice, il secondo come sensore delle colonne. In casi come questi però, la numerazione esadecimale non basta, occorre una transcodifica in qualche codice alfanumerico come l'ASCII. La soluzione prevede di solito una tabella, sita su EPROM, formata da 32 byte in ciascuno dei quali è il codice corrispondente al valore esadecimale del tasto (senza dimenticare il codice del blank corrispondente alla battuta della barra spaziatrice).

Supponiamo che questa tabella, ordinata in modo che nel primo, secondo... 32° byte sia contenuta la codifica del 1°, 2°....32° tasto, inizi all'indirizzo 00A0H. La ricerca tabellare inizia nel punto in cui la subroutine precedente aveva la prima RET (la modifica è illustrata nella stessa figura 9-10, a destra della detta RET).

La CALL RICTAB si inserisce nella codifica precdente come segue (al posto della prima RET C):

```
CICLIN  RLCA
        CALL C,RICTAB
        RET C
        DEC E eccetera...
```

Le due istruzioni centrali meritano un commento. *Entrambe* vengono ignorate se Cy = 0, in modo che in tale evenienza né si operi la ricerca in tabella né si esca prematuramente dalla routine. Naturalmente è necessario che entro la RICTAB il valore di Cy non venga modificato o, se ciò dovesse avvenire, si provveda opportunamente ad iniziarla con il salvataggio di AF ed a terminarla con il relativo recupero.

Senza questa, a nostro avviso, elegante procedura, si sarebbe più banalmente dovuto fare:

```
CICLIN  RLCA
        JR NC,AVANTI
        CALL RICTAB
        RET
AVANTI  DEC E ecc.
```

Vediamo infine RICTAB. Supponiamo sempre che il primo dato, corrispondente al primo tasto si trovi nella cella di indirizzo, come si è detto, A0H.

In testa al sorgente Assembly ci sarà lo pseudo codice:

TABCAR DEFM '. . . . . .'

con al posto dei puntini la rappresentazione, *in chiaro* dei contenuti del 1°, 2°, ... 32° carattere che automaticamente l'Assemblatore traduce in ASCII code. Quando, col meccanismo che abbiamo descritto e che implicitamente immaginiamo generalizzato a 32 chiavi, si va a chiamare RICTAB *l'accesso a TABCAR può farsi* molto semplicemente *in modo diretto:* senza cioè spazzolare l'intera tabella. Non si avrà difficoltà a rendersi conto come operano le seguenti istruzioni:

```
RICTAB      LD HL,TABCAR - 1      in HL è messo 009FH
            LD D,0
            ADD HL,DE             HL = 9FH + 1,2,...20H = 32D
            LD A,(HL)
            RET
```

Al termine del polling E = 3, corrispondente alla terza chiave, si ottiene HL = 9F + 3 = A2 che è l'indirizzo della terza cella di TABCAR.

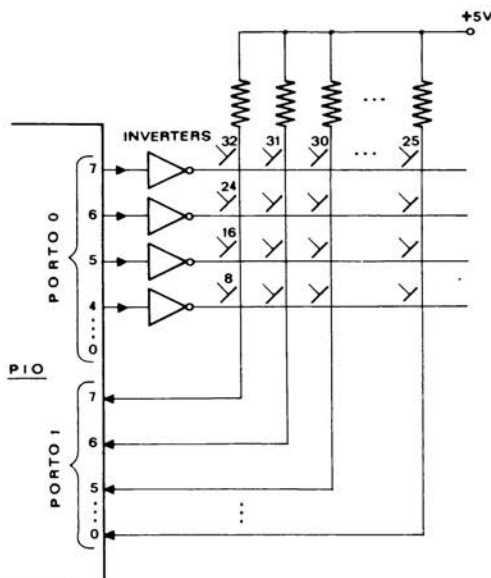



Figura 9-11 Tastiera 32 chiavi a matrice 4 x 8 (1 + 1/2 porti)

Programma P. 9-2

POLLING DI TASTIERA A 32 CARATTERI

Allo scopo di tirare le fila di un discorso che abbiamo dovuto fare a pezzi e bocconi, concludiamo con il sottoprogramma del polling di una tastiera a 32 chiavi. Come si può vedere nella figura 9-11 immaginiamo di utilizzare un porto 0 di una PIO o, meglio, i bit 7, 6, 5 e 4 di esso per le linee di output della scansione, mentre i fili di colonna (otto) sono collegati ad altrettanti bit del porto successivo, diciamo l'1. Immaginiamo poi che, come spesso avviene in casi del genere, come buffer per i quattro fili di linea vi siano altrettanti invertitori e che i fili di colonna siano connessi con resistenze al + 5V. In tal modo, quando un tasto è battuto o la linea cui appartiene è attivata, viene posta una massa, livello 0, al corrispondente bit di input del porto 1. Per ricondurci nelle stesse condizioni di prima, nel corso della scansione delle colonne (ciclo ora chiamato LITTLE), basterà la semplicissima istruzione CPL (assoluto 2F):

> Se ad es. è attivata la linea 6 ed è premuto il tasto 23 il dato letto sul porto 1 sarà: 10111111. Con l'istruzione CPL l'Accumulatore diventa: 01000000 ed è come se fosse stato trasmesso l'uno.

Il programma risulta in definitiva il seguente.

Programma P. 9-2

Indirizzi	Label	Cod. Assembly	Commenti
00A0	TABCAR	DEFM '012345 ecc.....'	(32 codici a piacere)
	....		
	....		
	POLK 32	LD E,32	inizia con 32° caratt.
	LLD	LD A,1	A = 00000001B
	BIG	RRCA	A = 10000000, poi 01000000 ecc.
		OUT (0),A	emetti su porto output (linee)
		EX AF,AF'	salva dato precedente
	LEGDEB	IN A,(1)	lettura con debouncing
		LD D,A	del porto 1 di input (colonne)
		CALL RIT	da 2 a 5 msec.
		IN A,(1)	
		CP D	
		JR NZ,LEGDEB	
		CPL	complementa Acc.
		LD B,8	(qui i bit sono *otto*!)
	LITTLE	RLCA	ruota sin. con A_7 in Cy

Indirizzi	Label	Cod. Assembly	Commenti
		CALL C,RICTAB	se Cy = 1 fa ricerca tabell.
		RET C	e rientra anche da POLK32
		DEC E	forse è la chiave 31, 30...
		DJNZ LITTLE	riprova con ciclo piccolo
		EX AF,AF'	riprendi valore salvato
		BIT 4,A	è stata sentita linea 4?
		JR Z,BIG	se NO riprendi con BIG (nuova linea)
		XOR A	se SI POLK32 dà: A = 0; Cy = 0
		RET	
	RICTAB	LD HL,TABCAR - 1	
		LD D,0	
		ADD HL,DE	
		LD A,(HL)	
		RET	

Si noti che ora, come è più che evidente, i quattro RLCA del caso precedente non sono più necessari.

Conclusioni

Il discorso sugli ascensori ci ha portato a definire certi aspetti dei problemi di input/output che si incontrano in tantissimi problemi del genere. Questo approccio forse insolito ma, secondo noi, didatticamente valido di partire da problemi quasi-inventati per andare gradualmente a definire aspetti tecnici spiccioli che, man mano che il discorso tenta di farsi più concreto, saltano fuori inevitabilmente, ha il pregio di una miglior comprensione di quanto non avviene, nei manuali esclusivamente tecnici in cui i discorsi sull'input/output vengono introdotti direttamente.

Avvertiamo che con questa trattazione siamo ben lungi dall'aver esaurito il problema (per esempio la scansione di tastiere in modo misto, hardware/software, che è forse il più usato è stata solamente accennata); anzi per certi aspetti il discorso risulta schematico e semplificato.

Ma il nostro scopo era soprattutto di introdurre i problemi in gioco, (senza appesantire troppo il discorso, ad es., con i bizantinismi delle tecniche di debouncing più elaborate): d'altronde non si deve dimenticare che esistono testi validissimi che trattano tali problemi minuti e che le Case costruttrici generalmente forniscono "packages" in materia, bell'e pronti.

Da ultimo speriamo che, con gli elementi sparsi che abbiamo fornito tra il capitolo 8 e il presente, chi lo desidera possa arrivare a trasformare la simulazione in un impianto praticamente realistico: anche senza cabina e mo-

tori, si potrà utilizzare una (o più PIO) per collegarvi in vari modi, scelti tra quelli descritti, pulsanti per imitare chiamate di piano (e/o prenotazioni) e tasti per simulare i sensori. Per il monitoraggio dei piani raggiunti si potrà o usare ancora l'unità video del NASCOM oppure dei LED collegati sempre ad un porto della PIO, o anche a dei latch indirizzati con la tecnica del memory mapping qui descritta.

Suggerimenti: a) 4 pulsanti e 4 interruttori per chiamate e sensori di un semplice "lift" di 4 piani, sul porto A della PIO; 4 LED per segnalare, magari via software, i piani raggiunti collegati al porto B, con i 4 bit rimasti liberi anch'essi connessi a LED per le uscite: SAL, DISC, ecc.;
b) *priority encoder 74148* da 8 linee in tre, per 8 piani: utilizzandone due, uno per i pulsanti, uno per i sensori, si hanno 3 + 3 bit del porto A e restano liberi due bit di questo più gli 8 eventuali di B per l'output ecc. ecc. ecc.

Anche il carro ponte in varie versioni può essere pure oggetto di svariate, non difficili implementazioni.

CAPITOLO 10

ALTRI ESEMPI APPLICATIVI IL CONTROLLO DI PROCESSO

In questo ultimo capitolo daremo dei cenni su due altre applicazioni di controllo a microprocessore. Queste come si sa sono numerosissime e tutte interessanti, vanno dai filtri digitali ai video games, dagli strumenti di misura intelligenti ai terminali per l'acquisizione dati, per non parlare di applicazioni industriali: economizzatori, dosatori nonchè quelle complesse macchine per il controllo di più punti di un impianto, in cui gli interrupt e le ghirlande di margherite ("daisy chain") si sprecano.

Ma a parte che, a voler parlare di tutte queste applicazioni, non basterebbe un'opera di 10 volumi in-folio, pensiamo che il nostro compito termini qui: volevamo solo aprire una finestra sul mondo affascinante dei microprocessori. Speriamo di essere riusciti ad invogliare a continuare lo studio (oltre che sui testi, sulle riviste specializzate e, soprattutto, per chi ne ha opportunità, nella progettazione reale) facendo capire che in questo settore c'è molto da soffrire, ma ci si può anche divertire.

Collaudo apparecchiature di serie

Si immagina che apparecchi di serie di una qualche complessità subiscano un test automatico su 100 punti di misura (si è scelto tale valore, corrispondente in esadecimale a 64 H solo perchè è un numero tondo). I 100 valori vengono rilevati da un apparato di misura automatico che è operato dal microprocessore, tramite una routine AVANZ che ci limiteremo a citare (ma che si può facilmente immaginare come funzioni, almeno nelle grandi linee, si pensi al carro ponte e simili). I valori previsti stanno in una tabella TABTST sita, per i motivi che stiamo per vedere, *in fondo* alla EPROM su cui risiede il programma.

Anzichè su EPROM, in certe applicazioni in cui si richiede maggiore flessibilità (esempio: macchina che collauda apparecchi dello stesso tipo ma con valori delle misure diversi) questa tabella risiederà su RAM.

Il flow chart panoramico, che quasi non meritava di essere disegnato talmente e ovvio, prevede il seguente ciclo:

-INIZIO — insieme di operazioni di installazione del sistema, variabile da caso a caso in quanto connesse al particolare hardware utilizzato. Saranno previste comunque le seguenti operazioni: inizializzazioni unità tipo P10 ecc. caricamento eventuale da cassetta su RAM

della detta tabella; controllo che certi sensori segnalino l'arrivo in stazione di misura dell'unita da collaudare (in loop d'attesa, con o senza indicazione di "time out" ecc.) ecc. Si può supporre che lo start verso queste istruzioni INIZIO venga dato al micro con il comando RS oppure, se è previsto anche un operatore umano, che il comando sia dato da questi con dei tasti di una keyboard.(°)

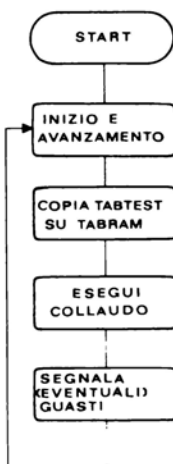



Figura 10-1

- COPIA della TABTTST — dei valori previsti in una TABRAM, posta questa immediatamente *a ridosso* dell'altra, nelle posizioni iniziali della memoria RAM; queste istruzioni si riferiscono, lo precisiamo, al particolare algoritmo che stiamo per illustrare (e che non abbiamo la minima pretesa costituisca un modello);

-ESECUZIONE — del collaudo tramite gli opportuni porti di I/O, gli eventuali convertitori Analogico/digitali ecc; in questa fase, ogni volta che si riscontra disparità tra

(°) - In casi del genere la macchina rassomiglia molto ad un sistema di sviluppo, con un monitor ecc.

valore misurato e valore previsto, quello misurato è caricato sulla TABRAM, nella cella corrispondente. Si provvede anche ad accendere un flag, per esempio Cy (siamo nell'ambito del detto algoritmo particolare);

-SEGNALAZIONE eventuali guasti: terminato il collaudo, si va ad interrogare Cy e, se esso è zero vuol dire che tutto andava bene e si segnala, tramite un'opportuna unità di output, dal più semplice display a LED ad un CRT o anche TTY (Teletype) questo (lieto) evento. In caso contrario, si esplorano le due tabelle e si emettono le coppie di valori corrispondenti di TABTST e TABRAM che non risultano concordanti.
Terminate queste operazioni, si riprende dall'inizio, generalmente non proprio all'inizio 0000 ma a quelle istruzioni che servono solo al posizionamento di una nuova macchina nella stazione di misura.

Le ultime parti possiamo ora vederle in linguaggio Assembly. Per una volta tanto, visto che il sottoprogramma AVANZ non lo possiamo definire nei dettagli in quanto si riferisce alle operazioni di passaggio a nuova misura, utilizzeremo la tecnica della MACRO che in un sistema di sviluppo industriale difficilmente può mancare.

Uso di una macro

Il nome della *macro* viene semplicemente inserito come se fosse un opcode, mentre le istruzioni che la compongono sono iniziate con una label identica al nome della macro e terminate, anzichè con la RET che invece contraddistingue le subroutine, con il pseudo opcode ENDM (= fine Macro).

Esempio: la nostra macro AVANZ avrà l'aspetto seguente, in fondo al programma (o anche all'inizio):

Label	Codice Assembly	Commenti
AVANZ	IN A, (2)	1ª istruzione (indicativa)
	. . . . .	altre
	. . . . .	istruzioni della MACRO
	ENDEM	fine MACRO.

Programma P. 10-1

COLLAUDO APPARECCHIATURE (parte centrale)

Dopo quanto abbiamo premesso, questa parte si presenterà come segue:

Label	Codice Assembly	Commenti
LUNGTAB	EQU 64H	la tabella ha 100 elementi
TABTST	EQU xxxxH	xxxx = indirizzo 1° elem.
TABRAM	EQU yyyyH	yyyy = " 1° elem. TABRAM
	XOR A	reset Carry
	LD HL, TABTST	
	LD DE, TABRAM	
	LD BC, LUNGTAB	
	LDIR	copia TABTST in TABRAM
	LD HL, TABTST	
	LD DE, TABRAM	
	LD C, LUNGTAB	(tanto, dopo LDIR, B =0...)
	MISURA	prima MACRO: fai misura
	CPI	confr. Acc. (= (misura) con (HL)
	CALL NZ,DIVERS	se discordi, esegui DIVERS
	INC DE	non implicata dalla CPI!
	AVANZ	seconda macro: altra misura
	LD A,C	
	OR A	set flag Z!
	JR NZ, MISURA	se C ancora ≠ 0, altra mis.
	. . . .	segue parte *segnalazione*.

Osservazioni: anzichè con una EQU, la TABTST *non* poteva essere definita mediante una DEFM, dato che questa carica dati ASCII e qui è implicito, per non avere limitazioni troppo serie nel campo dei valori, l'impiego di dati esadecimali (d'altronde, una volta che l'assemblatore ha fatto il suo lavoro, si sarebbe comunque da caricare, il solito a mano, il programma oggetto, la cui coda è rappresentata appunto dai valori di TABTST, comunque codificati; l'istruzione CPI (= ComPare & Increment) confronta automaticamente i successivi valori di TABTST con i dati delle misure che via via affluiscono, da un porto di input, in Accumulatore, quindi incrementa HL e decrementa BC. Quando i due dati differiscono si esegue la routine DIVERS che è del tipo:

```
DIVERS        SCF            Set Carry flag
              LD (DE), A     misura errata in (DE)
              RET
```

Si poteva forse evitare la chiamata, ma queste CALL condizionate sono così eleganti!...

Con la INC DE si passa ad un altro elemento della TABRAM, in modo che le due tabelle procedano di pari passo.

Prima di vedere come è fatto il resto, parliamo di alcuni modi in cui può essere fatta la MACRO che abbiamo denominato MISURA.

Supponiamo sia (0) il porto su cui arriva la misura effettuata.

Prima soluzione. Si realizza un hardware (non difficile da implementare), che *prima* dell'inizio della misura emette il valore "inesistente" FF (si suppone che, ovviamente, i valori effettivi anche errati siano sempre inferiori a 255, il che è naturalmente un limite sovente eccessivo). Questo valore, supponiamo venga mantenuto finchè la misura non è a punto.

Con queste premesse, non è difficile comprendere come opera la seguente routine:

```
MISURA      MACRC

            IN A, (0)
            CP FFH          è il valore "inesistente"?
            JR NZ, TEST     se NO, attendi tale segno
CALMA       CALL DELMIS     DELMIS = tempo minimo misura
            IN A, (0)       rileggi, per vedere se:
            CP FF           misure ultimate?
            JR Z, CALMA     non ancora,se c'è sempre FF
            CALL DELASS     ritardo assest.to valore
            IN A, (0)       leggi valore assestato
            ENDM            fine della MACRO.
```

Il ritardo DELMIS corrisponde ad un tempo minimo di misura, durante il quale cioè è inutile leggere il porto. Beninteso, si potrebbe anche eliminare questo ritardo (tanto lo Z80 non si spazientisce!...) ma in certi casi potrebbe avere forse una funzione simile a quella del "debouncing". Una tale funzione, ce l'ha più esplicitamente DELASS: anche se in questi problemi non si parla generalmente di contatti, un tempo di ritardo per l'assestamento del valore della misura è in generale necessario. In definitiva il segnale FF opera in questo modo: quando si presenta, denota l'inizio della misura; finchè si mantiene indica il perdurare di tale operazione e, quando cessa, avverte della fine di essa: una tecnica che, come si può comprendere, torna assai utile non solo in un caso come questo.

Seconda soluzione. Poichè la MACRO denominata AVANZ possiamo ritenere avrà probabilmente bisogno di un altro porto, diciamo (1), per un numero, di solito, relativamente esiguo di bit (persino uno soltanto) per l'emissione del comando all'apparato di misura (passare ad un altro punto di rilevazione). Si può così pensare che resti disponibile un bit di tale porto della PIO per la pre-segnalazione della misura. Supponendo allora che esso sia il bit 0 si avrà (pensando di togliere DELASS, forse eccessivo se si sup-

pone che quando il bit 0 torna a zero esso segnali, con sicurezza, misura stabile):

```
MISURA      IN A, (1)
                       BIT 0, A          c'è il "preallarme"?
                       JR Z, MISURA      se NO torna ad attenderlo
RIT                    CALL DELMIS       attenti il minimo ass.to
                       BIT 0,A           bit 0 è uguale a zero?
                       JR NZ, RIT        se non ancora, attendi
                       IN A, (0)         ora leggi mis. stabilizz.
                       RET
```

Naturalmente si potrebbe anche fare semplicemente il bit 0 segnali solo la fine della misura e/o fare DELMIS pari, anzi un pò superiore al tempo di assestamento della misura più lenta, ma è chiaro che il metodo visto dà maggior sicurezza. (°)

Un'altra possibilità potrebbe essere fornita dal bit, per uniformità immaginato sempre lo 0, definito come generatore di un interrupt. Immaginando quest'ultimo *vettorizzato*, al posto di MISURA si avrebbe semplicemente l'istruzione HALT. Sarà opportuno che essa sia immediatamente preceduta da una EI (o dalla analoga parola per la PIO), in modo che solo con l'HALT l'interrupt possa agire. Si può pensare poi che l'interrupt venga azionato semplicemente *alla fine* di una misura. In tal caso la routine dell'interrupt è banale: IN A, (0) seguita da RETI. In tal modo si va alla istruzione seguente la HALT e il programma prosegue.

In tutti gli esempi che precedono era inteso che la PIO operava in input/output mode (modo 3).

Un altro modo di generare l'interrupt consiste nel collegarsi direttamente alla linea INT della CPU. Immaginando stavolta di usare l'Interrupt Mode 1, che , lo ricordiamo, fa saltare automaticamente alla posizione 0038H, ci converrà modificare il MAIN come segue:

```
            ......
MISURA      HALT              attesa interrupt
            IN A, (0)         leggi la misura
            CPI
            CALL NZ,DIVERS
            resto come prima
```

(°) - Si noti come il fatto che la DEIMIS sia pari al tempo *minimo* di assestamento della misura più veloce garantisce un tempo *medio* di attesa pari allo stretto indispensabile.

Nella posizione di memoria di indirizzo 0038H sarà sufficiente porre semplicemente l'istruzione RET (non RETI, nota bene, essendo quest'ultima obbligatoria ma solo con l'IM2), sicchè il tutto *costa un solo byte* (e conosciamo dei perfezionisti funambolici che, per pura scommessa riescono a rendere addirittura nullo il costo, facendo in modo che una qualche altra routine - del monitor ad esempio - termini proprio in posizione 0038H!).

Tornando un momento alla tecnica impiegata nelle prime due soluzioni, i lettori più acuti si saranno chiesti: che cosa assicura che il segnale di "inizio misura" arrivi al punto (più o meno) giusto, in modo, soprattutto, che nessuna operazione di misura venga ignorata dal controller a microprocessore? La risposta è semplice: quel segnale non si produce in modo del tutto casuale ma *viene indirettamente, fatto partire dallo stesso microprocessore.* Ciò avviene nella parte introduttiva (in cui si torna ciclicamente dopo ogni serie delle supposte 100 rilevazioni) per la prima misura e, per tutte le altre, nel corso della MACRO denominata AVANZ:

il tutto marcia così: il micro dà l'avvio della 1ª, 2ª...ecc. misura (ponendo a zero il bit di segnalazione inizio); l'apparato esterno (magari ancora parzialmente sotto controllo dello Z80) si muove e compie operazioni preliminari, al termine del quale lancia il messaggio di "inizio misura"; il controller a micro legge la misura appena terminata e stabilizzata, poi dà l'avvio per una 2°, 3°...e così via, misurazione

Se ci si riflette, si tratta di un meccanismo che rientra nella denominazione di "handshake" che è meccanizzata nelle PIO, SIO, CTC. Pertanto un.altra possibiltà è la generazione di un interrupt tramite lo "strobe" della PIO.

Parte conclusiva: segnalazione

In questa parte, come in tutti quei problemi in cui si hanno in gioco *tabelle correlate* (cioè i cui elementi si corrispondono a due a due), può convenire l'uso di un registro indice, supponiamo di scegliere IX.

Spendiamo prima due parole circa l'uso dei registri indice. Va subito detto che troppi programmàtori, probabilmente perchè abituati in precedenza con microprocessori 8080, praticamente ignorano questi registri. La scusa che i relativi opcode hanno più byte di quelli corrispondenti con HL, DE e BC, non sempre regge: quando specialmente si debbono puntare più zone distinte di memoria operare solo con le coppie HL, DE BC rende necessari funanbolismo tali che vengono evitati, magari con risparmio complessivo di byte di programma, con l'uso si IX e IY e, oltretutto, ne guadagna enormemente la chiarezza.

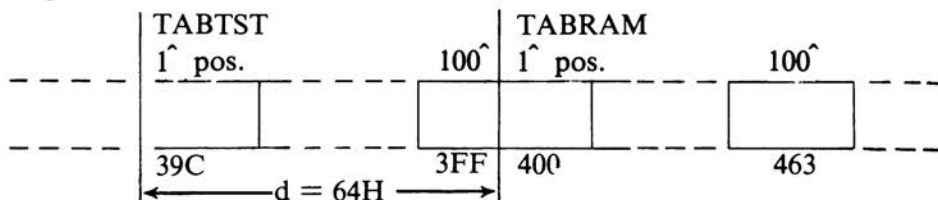
Un maggior uso di essi si avrebbe, probabilmente, se esistessero istruzioni che chiamano in gioco solo metà di IX o IY, così come avviene con le coppie HL ecc. Recentemente dei ricercatori hanno scoperto che esistono codici assoluti, diciamo così "non ufficiali" che si riferiscono alle sole metà destre e sinistre di IX e IY. Purtroppo gli assemblatori ignorano tali possibilità, lo sfruttamento delle quali non si presenta quindi pratico (per chi provi a guardare l'articolo comparso sul n.3 della rivista BIT, in un mese tra aprile e giugno 1979).

Con i registri indice si deve precisare anche, nelle istruzioni che li chiama in ballo, la *dislocazione* (o spiazzamento, ingl. displacement) *d.*

Nel nostro caso, con spiazzamento d = 0 si punterà ad un elemento della TABTST e con d pari alla distanza tra questo elemento ed il corrispettivo della TABRAM si punterà a quest'ultimo. Ora è evidente che, potendo essere d pari, al massimo,a 127 (esadecimale 7F), conviene sempre porre tabelle correlate, specie se abbastanza lunghe come nel nostro caso, l'una di seguito all'altra. Nel nostro caso sarà quindi:

$$d = LUNGTAB = 64H$$

Vediamo la cosa più da vicino. Abbiamo già detto che TABTST termina all'ultima posizione della EPROM. Sia questa di 1k; segue che il relativo indirizzo è: 0 3FFH = 1023D (si ricordi sempre che il primo indirizzo è 0000).

A chi segue queste note non sarà difficile trovare che il primo byte di TABTST è invece 39CH (cioè 400 - 64, in esadecimale), mentre il primo byte di TABRAM è il successivo di 3FF. La situazione è riepilogata qui di seguito:

In conclusione, il resto del programma è il seguente (con al termine il ritorno alla parte introduttiva ove si passa ad altro collaudo):

```
         ......
         CALL NC TUTTOK     se Cy = 0 segn. collaudo O.K.
         JR NC FONDO        e salta segnalaz. errori
         LD B LUNGTAB       B = lunghezza tebelle
         LD IX TABTST
SOPRA    LD A (IX + 0H)     punta a TABTST: poni in Acc
         CP (IX + 64H)      cfr con corrisp. el. TABRAM
         CALL NZ SEGNER     se diversi, segnala errore
         INC IX             altra coppia correlata
         DJNZ SOPRA         se ce n'è ancora...
FONDO    JP ALTCOLL         con B = 0, altro collaudo
```

Le routine TUTTOK (= TUTTO O.K.) e SEGNER hanno funzioni rispettivamente di display del collaudo superato e della segnalazione misure non normali, come già annunciato inizialmente.

Già che siamo in argomento, immaginiamo di aver a che fare con tabelle correlate molto lunghe. Precisiamo intanto che la massima distanza utilizzabile con IX e IY è, in modulo, 128. In tal caso, non si dimentichi, è necessario lavorare con spiazzamento d *negativo* (con d positivo si ha comegià detto un massimo di 127). Nel caso nostro poi non è necessario invertire l'ordine delle due tabelle (cosa che *non ci conviene proprio!); basterà utilizzare il displacement nullo per TABRAM e puntare a TABTST con d* = -128.

Se comunque si supera tale limite, è giocoforza tornare ai soliti puntatori HL e fratelli, dato che IX e IY usati sistematicamente con d = 0 non convengono in genere più, essendo più pesanti come byte impegnati (sempre, beninteso che non si preferisce tenere riservati ad altri scopi HL,DE e BC, senza girandole di PUSH, POP ed EXX).

Supponiamo, per esagerare di proposito, che si superi il numero di 256 elementi. Avremo bisogno,ad esempio di BC in funzione di "Byte Counter". Una possibile soluzione è, in queste ipotesi, la seguente:

```
                CALL NC TUTTOK
                JR NC, FONDO
                LD BC, LUNGTAB
                LD HL, TABTST      HL punta a TABTST
                LD DE, TABRAM      DE " " TABRAM
RIPCFR          LD A, (DE)         carica in Acc. el. to TABRAM
                CPI                cfr. elementi correlati
                CALL NZ SEGNER
                INC DE              non implicata da CPI
                JP PE, RIPCFR      se BC = 0 ripeti confronto
FONDO           JP ALTCOLL         se BC = 0, altro collaudo.
```

L'ultima istruzione prima di FONDO merita una spiegazione. Come si può controllare leggendo attentamente il manuale dello Z80; con la CPI la condizione BC = 0 è segnalata dal flag P/V (Parity/oVerflow) il cui stato è evocato dal codice condizione PE (Parity Even) corrispondente a detto flag = 1 (tale di mantiene, con la CPI e simile finchè BC≠0), mentre con BC = 0 si ha la condizione PO (ossia Parity Odd, flag P/V =0). Il motivo di questa scelta devrebbe essere chiaro, dal momento che il flag Z è qui riservato alla comparazione.

(°) - Come pure con le istruzioni CPD, CPIR, CPDR, nonchè le analoghe di trasferimento blocchi: LDI, LDD, LDIR e LDDR.

Un'ultima osservazione: le istruzioni di inizializzazione di HL e di DE, volendo, si sarebbero potute evitare (così facendo però il procedimento risulta più chiaro): dato che nella prima parte del MAIN program i due puntatori erano arrivati al fondo delle due tabelle, basterà utilizzare l'istruzione CPD, ossia Compare & Decrement in lungo della CPI. Sempreché beninteso il fatto che il confronto viene fatto a ritroso non crei inconvenienti nella segnalazione dei valori errati....

Attenzione poiché prima di iniziare occorre raggiustare il valore di DE con una istruzione di DEC DE, dato che la CPI, con l'arrivo di BC a zero *non fa un ulteriore incremento di HL*, in modo che si termini con l'ultimo elemento di tabella. L'istruzione DEC DE può poi essere opportunamente posta entro il loop.

Tornando all'ultima versione che tutto sommato, anche in base alle ultime considerazioni appare la più "straigh forward" (come dicono i testi americani) la subroutine SEGNER potrà, con una procedura facilmente intuibile, emettere su display (magari identico al tipo TV con cui ci siamo abituati a giocare) i dati seguenti: numero della misura, *desunto dal valore di BC*; valore previsto, corrispondente all'elemento puntato da HL (oppure da IX, con dislocazione d nulla) e infine valore misurato, discorde, sulla base del puntatore DE (oppure dell'indice IX con spiazzamento 64H).

Il procedimento che abbiamo descritto, che, al solito, non rappresenta di certo un modello, pur impegnando un poco di memoria ci appare nel complesso abbastanza chiaro e facilmente comprensibile.

Misure entro limiti di tolleranza

In quanto precede c'è, a dire il vero, una approssimazione che in molti casi concreti (sono sempre esclusi i casi delle attrezzature interamente digitali, crediamo) è del tutto irrealistica: la pretesa cioè che le misure siano esattamente identiche a quelle previste.

La variante non è difficile. Pensiamo di utilizzare un registro indice, IX, per puntare a due tabelle correlate, chiamiamole TABINF e TABSUP, su EPROM, contenenti negli elementi corrispondenti i limiti inferiori e superiori di tolleranza di ciascuna misura. Facciamo poi un'ulteriore semplificazione, consistente nel mettere in atto la routine che abbiamo denominato SEGNER al termine stesso di ogni misura. Non si pensi comunque che il procedimento iniziale sia da buttar via: esso torna utile quando le misure viaggiano ad una certa velocità, mentre il dispositivo di output è piuttosto lento. Comunque a questo punto coloro che hanno avuto cuore di arrivare fin qui saranno vaccinati contro il nostro metodo di partire con versioni iniziali relativamente imperfette o errate addirittura. Essendo costoro ormai anche piuttosto bravi, non vogliamo tediarli con ulteriori commenti a proposito della seguente versione.

Programma P. 10-2

COLLAUDO APPARECCHIATURE (misure entro limiti)

Label	Codice Assembly	Commenti
LUNG	EQU 64H	lunghezza tabelle
TABINF	EQU xxxxH	limiti infer. di tolleranza
TABSUP	EQU TABINF+LUNG	al termine EPROM: limiti sup
TABMIS	EQU TABSUP+LUNG	all'inizio RAM: valori mis.
	ORG	
INIMAIN	INIZIO	macro per inizializzazioni
	AVANZA	macro di avanz.to misura
	XOR A	reset Carry
	LD IX,TABINF	punta a TABINF (e TABSUP)
	LD HL,TABMIS	
	LD B,LUNG	
	MISURA	macro: misura in Acc.
	LD (HL),A	v. nota 1
	CP (IX + 0)	cfr. con TABINF
	CALL M,SEGNER	se minore, segnala errore
	CP (IX + LUNG)	cfr. con TABSUP
	CALL P,SEGNER	se maggiore, segn. errore
	INC IX	
	INC HL	
	DJNZ MISURA	
	CALL NC, TUTTOK	v. nota 2
	JP AVANZA	nuovo collaudo
	. . . .	seguono MACRO e suboutine

Note: 1) HL (e TABMIS!) può essere eliminato; TABMIS potrebbe però essere utile per controlli complessivi (a richiesta dell'operatore: istruzioni relative in AVANZA); 2)Il set di Cy è naturalmente da inserire nella soubroutine SEGNER.

Variante funambolica

Per gli amanti del brivido software proponiamo un ulteriore esercizio che coloro i quali a questo punto siano stufi di tabelle e collaudi sono padroni di saltare (Signore perdonali perchè non sanno quello che fanno).

Anche se dopo quanto abbiamo appena visto si può pensare che una tabella su RAM non serva, immaginiamo ugualmente che in una qualche applicazione torni utile una tale metodologia. Tornando per semplicità ad immaginare una sola tabella di riferimento (anzichè TABTST, per dare a quanto segue un carattere più generico) vogliamo anche supporre che ora non interessi sapere quanto sia grande ciascun eventuale errore, ma solo se una certa operazione di input è risultata o meno *in accordo* con la corrispondente della TABRIF. Anzichè allora una tabella di byte un autentico.....genovese ha pensato di utilizzarne una di bit! Il valore 1 indicherà "misura

valida" lo 0 errore. Siano ora 128 gli elementi della TABRIF, la TABRAM sarà allora di: 128 : 8 = 16 byte. Le coppie di elementi si corrisponderanno allora come illustrato nella seguente figura:

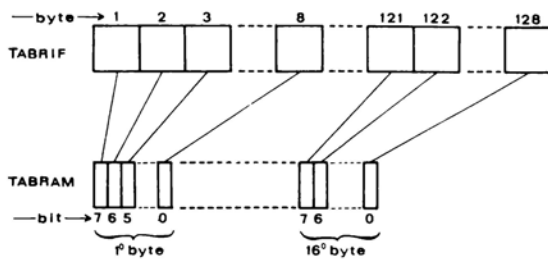



Figura 10-2 Correlazioni byte-bit.

La soluzione è la seguente:

Label	Codice Assembly	Commenti
LUNG	EQU 80H	lunghezza TABRIF
TABRIF	EQU xxxxH	
TABRAM	EQU TABRIF+LUNG	
	XOR A	solito Cy=0
	LD HL,TABRAM	
	LD B,16	16=n° elem. TABRAM
VALID	LD (HL),FFH	riempimento di "uni"
	INC HL	dei bit di
	DJNZ VALID	TABRAM
	LD HL, TABRIF	HL punta ai riferimenti
	LD DE,TABRAM	DE alle misure
	LD C,LUNG	
	LD B,8	n° bit di un byte in B
	MISURA	solita macro
	CPI	
	EX DE,HL	scambia puntatori
	JR Z,SOT	se misura valida, va SOT
	SCF	set Carry Flag, se errata
	RES 7,(HL)	bit errato = 0
SOT	RLC (HL)	ruota il bit
	EX DE,HL	ripristina vecchi DE e HL
	DJNZ AVANZ	Decr. B, salta GIU se non O
	LD B,8	altrim B=8
	INC DE	punta ad altro byte TABRAM
GIU	......	resto del
	......	programma (cfr. P. 10-1)

Vediamo un pò come gira il tutto. Al posto della copia di TABTST in TABRAM abbiamo ora semplicemente il riempimento di quest'ultima con valori FF, ossia con degli "1", indicanti *validità assunta"* (con aprioristico ottimismo, salvo verifica) delle varie rivelazioni. Quindi, dopo i soliti puntamenti di HL e DE, viene messa in C la LUNGhezza di TABRIF ed in B il valore 8.

Infatti B ora serve come de-contatore nella routine che inizia con EX DE,HL dopo la CPI. Ciò ha lo scopo di passare successivamente dal bit 7 al bit 6 ecc. fino al bit 0 di ogni byte di TABRAM. Ora nasce un problema che poi si rivela falso: poichè l'istruzione CPI implica il decremento della coppia BC, non c'è mica il pericolo che venga influenzato anche B? La risposta è *negativa*: infatti il decremento di BC implicato dalla CPI, come si vede in P.10-1 ha termine quando C = 0 e, fino ad allora, B non è toccato (anche qui, nella parte che segue, ci sono le istruzioni LD A,C poi OR A e JR NZ, MISURA .....).

Commentiamo infine le istruzioni che seguono la CPI.

Si inizia con lo scambio tra DE e HL. Questo allo scopo di usare come puntatore HL, anzichè DE, dal momento che le due istruzioni successive, RES 7, (HL) e RLC (HL) *funzionano solo con questo puntatore*. Le due citate istruzioni servono, la prima, ad azzerare il bit 7 della posizione di memoria precedentemente puntata da DE, ora da HL, mentre l'altra routa il bit 6, 5,....0. Se la misura è discorde da TABRIF, oltre al RES 7 (HL) viene fatto il solito set di Cy; invece la rotazione del bit è effettuata in ogni caso. Ricordando che alla label MISUR si ritorna ciclicamente per altre 127 volte, non dovrebbe essere difficile comprendere che il meccanismo "gira" come si deve, tenendo anche conto delle istruzioni successive: dopo lo scambio di DE ed HL, che rimette i puntatori in condizioni di guardare alle solite tabelle, si decrementa B in modo che, quando ogni 8 passi esso va a zero, lo si riporta al valore 8 e si passa, con INC DE, al byte successivo di TABRAM.

La parte relativa all'evidenziazione in output delle misure in disaccordo dovrà fornire una lista comprendente in ogni riga: il numero della misura fasulla (corrisponde all'indirizzo della TABRIF correlato al bit che risulta nullo) e (solo opzionalmente) il valore contenuto in tale cella di TABRIF.

Al lettore volonteroso questo sano divertimento.

Terminiamo accennando ad un'altra applicazione messa in atto da tecnici di nostra conoscenza, che ha una certa parentela con quella che abbiamo finora discusso.

Si tratta di un dispositivo economizzatore per il comando di un impianto di verniciatura di superfici irregolari poste su di un piano.

Nella prima fase veniva fatto il rilievo delle quote minime e massime entro le quali era compresa la superficie (per un numero di coppie sufficienti ai successivi viaggi alternativi dello spruzzatore). I dati così rilevati erano

caricati in due tabelle su RAM, tra loro, evidentemente, correlate. Nella seconda fase il microprocessore comandava il viaggio entro quelle quote. È un altro possibile esercizio, ancora più sano e dilettevole.

Controllo di processo

Generalità

Passiamo ora ad una applicazione che riveste la massima importanza tra quelle cui il microprocessore è dedicato, diremmo anzi che si tratta della più tipica di esse: il *controllo di processo*. Qui di seguito ne parleremo tuttavia nelle linee essenziali, ma non ce ne vergognamo dato che una trattazione esaustiva richiederebbe un volume monografico.

Riferiamoci al caso più semplice, in cui si deve mantenere costante ad un valore prestabilito una certa grandezza fisica (pressione, temperatura, livello di un serbatoio, velocità di una macchina operatrice ecceterà).

Come dovrebbe essere noto a chi sa qualcosa di elettronica industriale, nei sistemi di controllo automatico di tipo *analogico* l'impianto funziona come si suol dire *ad anello chiuso*: la grandezza analogica di comando, detta anche *riferimento* è confrontata con un segnale proveniente dal puntó finale della catena, cioè dall'uscita. Questo segnale di ritorno o di *feedback* è di solito fornito da un dispositivo che converte le dimensioni fisiche della grandezza controllata quando come quasi sempre avviene non è elettrica, in quelle di una tensione (o corrente) in modo che il confronto con il riferimento si possa avere. Questo dispositivo prende il nome di *trasduttore*, che in sostanza funge un pò da strumento di misura dell'uscita.

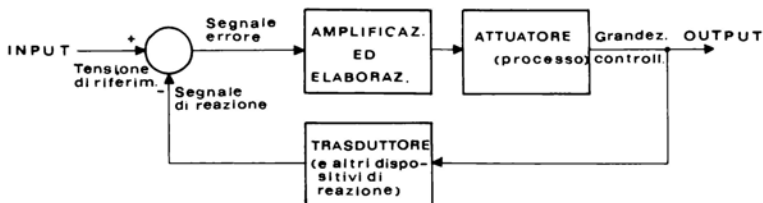



Figura 10-3 Schema di un anello di regolazione.

In base all'*errore*, cioè alla concordanza o meno tra il riferimento e il segnale *di reazione* (sinonimo di feedback) il resto della catena, formata solitamente, nella parte iniziale da reti attive con operazionali, (°) provvede

---

(°) - Ma, avvertiamo, si danno anche controlli automatici formati da elementi di tipo pneumatico, del tutto insensibile al "noise" elettrico.

all'amplificatore e alla elaborazione (analogica) del segnale errore in modo da correggere eventuali deviazioni dell'uscita dal valore prefissato con il riferimento. Questo schema di regolazione è illustrato nella figura 10-3 ove il cerchietto con le frecce confluenti con i segni + e - indica confronto tra riferimento e tensione di reazione, mentre il blocco denominato *attuatore* schematizza ogni dispositivo atto a trasformare il segnale elaborato nella grandezza che è oggetto del controllo (si tratterà, a seconda dei casi di servomotori, dispositivi riscaldanti, valvole eccetera).

Facendo ora dei brevissimi cenni storici, il controllo di processo *di tipo digitale* si è cominciato a realizzarlo nei grossi impianti (dove gli anelli di feedback sono molti, complessi e tra di loro collegati) mediante grossi elaboratori studiati appositamente per tali applicazioni e detti appunto elaboratori di processo. In seguito, parliamo della decade che precede gli anni 74-75, c'è stato l'avvento dei *minicomputer* che, con i loro costi minori (e le loro caratteristiche specifiche, naturalmente: vedi gli interrupt) hanno diffuso il controllo di tipo numerico anche sugli impianti di media complessità.

Mentre si andava anche sviluppando, naturalmente, la tecnica del controllo digitale a logica cablata (l'esempio più importante è costituito dalle macchine utensili a controllo numerico) con l'avvento dei microprocessori e dei microcomputer (la cui potenza elaborativa emula ormai quella dei mini, al punto che tutte queste distinzioni ormai sembra perdano senso) le applicazioni di controllo mediante elaboratore numerico si vanno estendendo con legge esponenziale.

Una semplicissima scheda per il controllo di processo

Nei controlli numerici le grandezze in gioco sono numeri codificati in binario, ad eccezione quasi sempre della grandezza di uscita. Pertanto anche in questi casi si attua il controllo o, come anche si dice, la *regolazione,* a ciclo chiuso.

> Nel caso particolare di *attuatori numerici* come i motori passo-passo sono pure possibili realizzazioni ad anello aperto, ma anche qui, nei casi in cui si vuole un maggiore affidamento, è spesso consigliabile l'uso del feedback.

Occorreranno allora, almeno in generale, dei *convertitori* di due tipi:

- *convertitori Analogico/Digitali* (abbr. A/D) per trasformare la grandezza controllata, di tipo analogico, in un numero in codice binario (solo per certe grandezze come la posizione si hanno degli "encoder" che danno direttamente uscita numerica);

- *convertitori Digitali/Analogici* (abbr. D/A converter), per compiere l'operazione opposta: essi hanno in input il segnale di correzione proveniente dal microprocessore e lo trasformano, in uscita, nel corrispondente segnale analogico da inviare al processo (tramite eventuale amplificazione e filtraggio: un pò di elettronica analogica rimane sempre!)

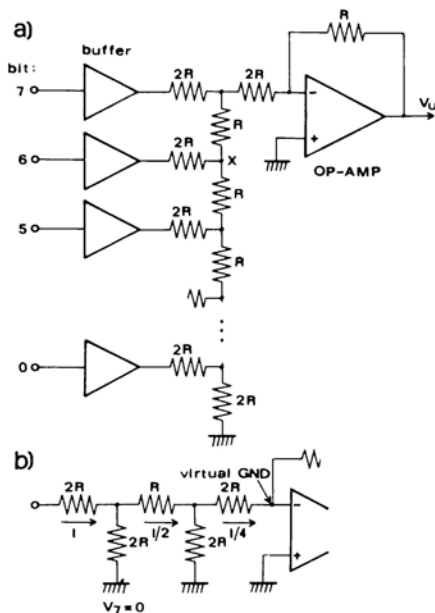



Figura 10-4 Convertitore D/A (DAC) con rete a scala R-2R.

Il micro serve allora a chiudere il loop, operando il confronto tra la grandezza di riferimento che esso tiene in memoria (finchè dall'esterno non gli provenga un ordine di modificarne il valore; può anche darsi il caso di un valore impostato sull'esterno di cui il micro fa periodicamente il "polling") e quella fornita dal convertitore A/D (e dal trasduttore che lo precede, naturalmente). In base al risultato di questo confronto ed eventualmente alla sua *dinamica* (il modo cioè in cui l'errore evolve nel tempo) il calcolatore di processo produce un segnale da inviare al sistema controllato (tramite il convertitore D/A) secondo procedure elaborative più o meno sofisticate (si parla di azioni di tipo "integrale","derivativo" o misto, mentre la più semplice è detta di tipo "proporzionale", tutte teorie che esulano abbondantemente dai limiti della presente trattazione).

Si comprende abbastanza facilmente anche da questi pochi cenni l'importanza che assumono in queste applicazioni i convertitori A/D a D/A abbreviati anche con le sigle ADC (= A/D Converter) e DAC (cioè D/A Conv.) che vengono ormai prodotti in forma integrata, con prestazioni sempre migliori e rapporto prestazioni/costo sempre maggiore. In genere, dei due è il tipo ADC il più economico, dal momento che, ad esempio, esso potrebbe essere implementato come nella figura 10-4a per mezzo di un operazionale e dei buffer sugli ingressi di una rete detta "*a scala*", anche senza ricorrere all'equivalente circuito integrato (più preciso, meno ingombrante e costoso).

Il funzionamento della rete a scala può essere riassunto dalla sua proprietà di mostrare, con degli ingressi in cortocircuito, in tutti i punti come quello segnato con X nella figura una resistenza pari a R se si guarda verso il basso e 2R se si guarda verso l'alto. Sulla base di questa considerazione, immaginando ora agente un solo bit alla volta, con gli altri nulli e tenendo presente che all'ingresso inverting dell'op-amp c'è una massa virtuale, con un po' di pazienza non sarà difficile rendersi conto che se il solo bit 7 è a livello alto e si chiama I la corrente che eroga il relativo buffer sulla 2R in alto a sinistra questa entra in un nodo sul quale si vedono due resistenze 2R in parallelo; pertanto sulla 2R di input dell'operazionale giunge una corrente I/2. Immaginando poi nullo il bit 7 e attivo il bit 6 la situazione è quella illustrata in b) nella medesima figura, in cui è immediato constatare che il contributo del bit 6 alla corrente in arrivo al "virtual ground" è pari a I/4. Estendendo il ragionamento ed applicando il principio di sovrapposizione è facile capire che la corrente di arrivo all'operazionale è la somma di contributi ciascuno corrispondente alla metà di quello del bit immediatamente più alto, quindi questa corrente è un equivalente analogico del valore binario di entrata (°), lo stesso si dica della tensione di uscita V_u che è proporzionale tramite la R_t di reazione alla detta corrente.

Conversione A/D software

Quanto alla conversione Analogico/Digitale, essa può essere anche effettuata anziché con il circuito ADC, più economicamente dal microprocessore stesso, impiegando un convertitore DAC e, come algoritmo, il *metodo delle approssimazioni successive.* Questo metodo (che è anche alla base di una classe di dispositivi a logica cablata ADC, del tipo più lento) è implementabile con del firmware applicato al micro. Esso funziona così:

Noto il campo dei valori della tensione di uscita V_{tr} del trasduttore (o altro), si deve far corrispondere questo valore limite al valore binario massimo a 8 bit, tanti quanti ne tratta il nostro Z80, come pure i suoi simili.

(°) - I buffer, oltre che assolvere a funzioni di isolamento e impendance matching debbono assicurare in uscita un'uguale corrente I a tutti gli ingressi della rete a scala.

Si imposta un primo tentativo basato sulla supposizione iniziale che la misura sia pari a metà del valore suddetto. Ciò equivale a porre il bit più pesante, l'MSB, ad 1. Si invia questo dato, cioè 10000000, al convertitore DAC, la cui uscita è confrontata in un comparatore con la tensione incognita. Il comparatore, attuato con un semplice operazionale, va alto quando V_i risulta maggiore della V_c che esce dal convertitore DAC (v. figura 10-5). In questo caso il bit MSB è confermato come "1", in caso contrario è posto a zero.

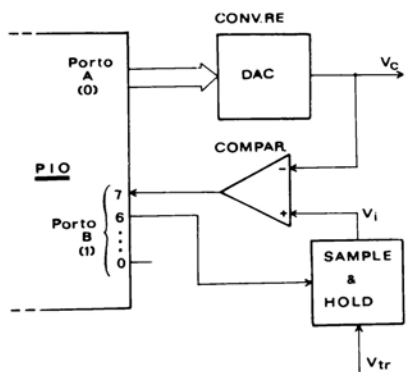



Figura 10-5 Schema per la conversione software A/D per approssimazioni successive.

Si passa allora al bit immediatamente inferiore e lo si pone a 1 a titolo di prova e lasciando naturalmente immutato il bit precedente, ormai affidabile, si invia la nuova approssimazione al DAC e si conferma o meno il nuovo bit. Il procedimento: ipotesi, invio dell'approssimazione al DAC, conferma o meno del nuovo "1", viene ripetuto fino all'esaurimento degli otto bit. Al termine, nel registro in cui sono stati successivamente impostati e provati i vari bit, ci sarà il valore binario equivalente alla V_{tr}. Immaginando sempre di utilizzare una PIO per I/O, si avrà il porto A come uscita dal micro ed entrata al DAC, mentre il porto B, operante in modo 3 avrà ad esempio sul bit 7 l'uscita del comparatore, come input e come output impiegherà il bit 6 per comandare un circuito di *Sample and Hold* che, specialmente in un procedimento di conversione alquanto lento come questo è indispensabile, affinchè la V_i non fluttui durante la conversione, ma si mantenga costante al valore ottenuto al momento della campionatura (fissato, appunto, da un impulso uscente dal bit 6 del porto B).

Sono evidenti i limiti del metodo:

1) anzitutto la lentezza, parzialmente ovviabile con un clock operante alla massima frequenza possibile (per lo Z80 si tratta di 4 Megahertz); questo

fattore è tuttavia di scarsissimo rilievo quando si ha a che fare con grandezze molto lentamente variabili (esempio: misure di tensioni continue oppure la temperatura di un grosso forno);

2) la risoluzione conseguibile con soli 8 bit è piuttosto scarsa; non volendo passare a micro a 12 o 16 bit, occorrerebbe complicare alquanto sia l'hardware che il software, a scapito sia della semplicità del tutto che della sua già mediocre velocità.

A proposito di quest'ultima va anche ribadita l'opportunità di un circuito Sample & Hold, che campiona ad un certo istante la V_{tr} e mantiene V_i a tale valore sino alla campionatura seguente.

Programma P. 10-3

CONVERSIONE ANALOGICO/DIGITALE

Il flow chart è nella figura 10-6. Inizialmente si provvede all'attivazione del Sample & Hold. A tale scopo, si invia un impulso brevissimo sulla linea 6

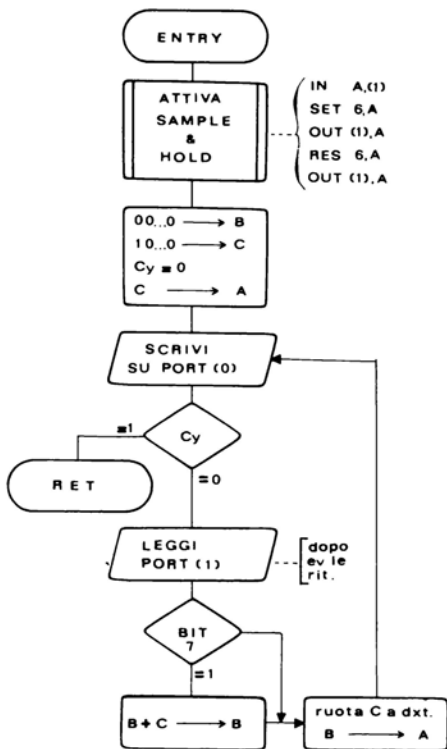



Figura 10-6 Flow chart della conversione A/D via software.

del porto B, immaginato indirizzato con (1): IN A,(1) seguita dal set del bit 6 e dall'OUT (1),A servono a lasciare immutati i bit diversi dal bit 6; poi il RES del bit 6 e la sua emissione hanno l'evidente scopo di far tornare a zero il brevissimo impulso (che dura quanto le istruzioni RES 6, A e OUT (1), A all'incirca; se il Sample & Hold scelto dovesse aver bisogno di impulsi più lunghi si può naturalmente inserire un ritardino prima di RES 6,A).

Si impiegano due registri, B e C, il primo destinato, al termine, a contenere l'equivalente binario di V_i, il secondo contenente il bit 1 di test che viene fatto ruotare successivamente dal MSB a bit via via meno significativi, fino al LSB. Dopo otto rotazioni, alla nona il bit unitario fa capolino nel Carry e questo serve da segnale per terminare la routine.

All'inizio, si ha B = 0 e C = 10000000B e questo valore è emesso sul porto (0). Quindi si legge (1) e si fa il test del bit 7: se esso è nullo, ciò sta a significare che V_i è minore di V_c (si osservi come sono disposte le entrate V_i e V_c del comparatore) e si salta alla rotazione senza assumere uguale a "1" il bit di prova, altrimenti questo viene inserito dal registro C nella corrispondente posizione di B. Esempio: al quarto giro sia B = xyz00000, con xyz valori di bit già collaudati nei precedenti tre giri; se il comparatore conferma il quarto bit "1", si fa l'operazione B = B + C cioè xyz00000 + 00010000 che fornisce B = xyz10000, altrimenti il quarto bit di B è confermato come nullo. Si noterà come nella codifica Assembly si sia preferito porre in gioco l'OR logico tra B e C, anziché l'operazione aritmetica ADD: il risultato è ovviamente il medesimo ma, ci si perdoni la sottigliezza, il significato ci è parso più pregnante, dato che il problema non è tanto dell'aggiungere un numero, bensì quello di "incastrare" un bit in una certa posizione binaria.

Segue, come già anticipato, la rotazione destra di un bit del registro C dopo di che si trasferisce B in A e questo sul porto (0), quindi si guarda Cy: se Cy = 1 il processo ha fine, altrimenti si ricomincia.

Dopo che il DAC ha fatto la sua conversine ed il comparatore è scattato in un senso o nell'altro, si potrà stabilire qual è il nuovo bit eccetera.

Perchè l'istruzione IN A, (1) possa essere inpunemente posta subito dopo la OUT (0),A è necessario che il tempo di transito dei segnali sul DAC e sul S. & H. sia convenientemente *minore* di quello necessario per l'espletamento della IN A, (1) medesima, ossia, con un clock che viaggi a 2 MHz, circa 5,5 microsec. tempo che dovrebbe essere più che sufficiente se si impiegano degli operazionali di media velocità. In casi meno favorevoli non si dimentichi mai questo problema, il cui rimedio è semplice: un opportuno ritardo perchè si abbia l'assestamento del fenomeno.

Codifica Assembly

Dopo il flow chart appena discusso, questa non presenta difficoltà di sorta.

Indir.	Contenuto	Label	Codice Assembly	Commenti
			ORG D00H	
D00	DB 01	ADCONV	IN A,(1)	manda un
D02	CB F7		SET 6,A	impulso
D04	D3 01		OUT (1),A	al
D06	CB B7		RES 6,A	circuito
D08	D3 01		OUT (1),A	Sample & Hold
D0A	06 00		LD B,0	B = reg. risultato conv.ne
D0C	0E 80		LD C,80H	C = 10000000B
D0E	AF		XOR A	reset Cy (°)
D0F	79		LD A,C	
D10	D3 00	PROVA	OUT (0),A	emetti "prova"
D12	D8		RET C	rientra se Cy = 1
D13	DB 01		IN A,(1)	leggi l'esito
D15	CB 7F		BIT 7,A	del comparatore:
D17	20 03		JR Z,ROTAZ	se bit 7 = 0, $V_i < V_c$
D19	78		LD A,B	altrimenti, con $V_i < V_c$
D1A	B1		OR C	"incastra" il bit 1
D1B	47		LD B,A	nel registro B
D1C	CB 01	ROTAZ	RRC C	ruota il bit del registro C
D1E	78		LD A,B	B in A, poi fa un (°°)
D1F	18 EF		JR PROVA	altro giretto di prova.

Per come sono state impostate le operazioni, si potrà notare che, al rientro da questa subroutine si ha il risultato della conversione sia in B che in A e, inoltre, anche nel registro di uscita del porto 0.

La scheda economica per il controllo di processo

Nella figura 10-7 è disegnato lo schema di una possibile scheda di basso costo atta all'interfacciamento tra il microprocessore e un processo di cui una sola grandezza è controllata. Per quanto già notato, poichè la conversione per approssimazioni successive non è delle più veloci, si supporrà trattarsi di un impianto di regolazione funzionante con forte inerzia(meccanica, termica) sicchè l'uscita controllata, destinata peraltro a mantenersi costante

(°) - Per prudenza, qualora di entri dal MAIN con Cy = 1.

(°°) - Dato che è A=B questa istr. è eliminabile (modificando la successiva da 18 EF a 18 F0!).

Figura 10-7 Scheda low cost per un controllo di processo (una grandezza).

per lunghi periodi, anche nei transitori varia comunque con grande lentezza.

Il trucco consiste nell'utilizzare il convertitore DAC per due funzioni: quella già vista per la conversione opposta ADC, via software e quella....di convertitore DAC che gli è propria.

Alla scheda affluisce il segnale V_{tr} proveniente dal blocco trasduttore (corredato quest'ultimo solitamente da un attenuatore tarato di valore opportuno, affinchè si stia nel range proprio del DAC). La V_{tr} entra in un buffer inseguitore (opzionale) la cui uscita è connessa all'entrata non invertente del comparatore tramite uno switch analogico. Questo è messo "on" (cioè la V_{tr} è lasciata passare) tramite il comando uscente dal bit 6 del porto B dell'unità PIO (funzionante in modo3). All'uscita dell'analogo switch c'è, in parallelo, un condensatore C', in modo che il tutto realizzi un semplice dispositivo Sample & Hold. Il comando uscente dal bit 6 dovrà allora essere impulsivo.

Un analogo switch è pure connesso in uscita dal DAC. Questo commutatore elettronico funziona ora come una valvola a due vie, mutuamente escludentisi: quando il comando uscente dal bit 5 del solito porto (1) è alto viene "apertà" la via superiore (e chiusa l'altra) onde la V_{DAC} è collegata al condensatore C" che così realizza un altro elementare Sample & Hold che, tramite un altro buffer, invia al processo un segnale di correzione elaborato dal microprocessore. Quando invece il bit 5 è basso la V_{DAC} non è più connessa all'uscita ma (è ora aperta la via superiore) all'input invertente del comparatore. Questo avverrà quando si opera la conversione software A/D e si noti bene l'importanza del fatto che l'analogo switch 1 disaccoppi la tensione V_{DAC} dalla V_{out}, dato che, durante la conversione la prima varia continuamente, mentre la seconda è bene si mantenga costante all'ultimo valore che è stato inviato al processo.

Le altre linee del porto B possono utilizzarsi per altri scopi ausiliari: allarmi, sensori particolari ecc. Qui si è pensato ad un gruppo di latch che pilotano dei LED per un semplicissimo dispositivo di *read-out* della misura effettuata, al termine della conversione A/D.Il circuito di read-out, opzionale, è comandato per mezzo del bit 4 (si può anche pensare alla tecnica del memory mapping naturalmente).

Flow chart generale di un controllo

Il diagramma a blocchi di massima per un regolatore utilizzante la detta cartolina (°) è illustrato nella figura 10-8.

(°) - Sulla cui validità in generale non facciamo la benchè minima propaganda. Essa è però utile didatticamente, perchè racchiude più funzioni.

Dopo le necessarie inizializzazioni della PIO, dello Stack Pointer e simili, si avranno opportune istruzioni per la messa in marcia dell'impianto, dopo di che inizierà il loop continuamente chiuso su se stesso, tranne l'evenienza di un interrupt di allarme o, se previsto, si un comando di arresto per fine lavoro.

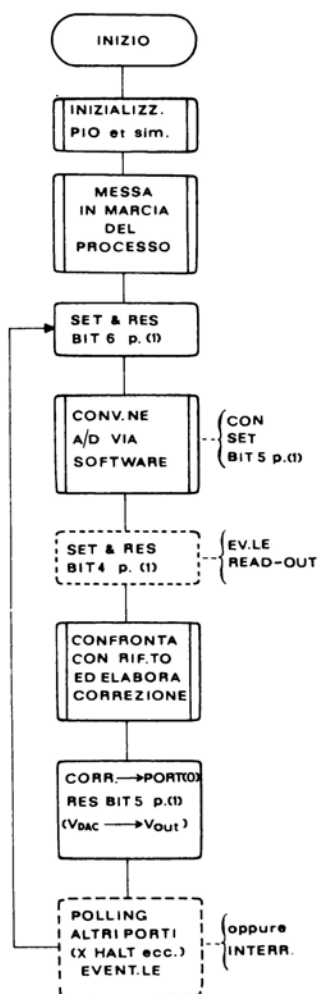



Figura 10-8 Diagramma generale di un controllo di temperatura (o simili).

Questo loop inizierà con il SET poi RES del bit 6, porto 1, che ha lo scopo di porre in C un campione del valore attuale di V_{tr}. Segue la conversione software già vista di questo dato analogico. Questa routine, come precisato nella nota a fianco, sarà preceduta dal set del bit 5 per connettere l'uscita V_{DAC} al comparatore e toglierla da C", come già precisato.

Dopo il set e reset del bit 4 per l'eventuale read-out della conversione, segue la parte centrale, il cuore del programma, in cui si elabora secondo algoritmi più o meno sofisticati la correzione. Questa è poi inviata sul porto 0 e, con il reset del bit 5, porto 1, questa correzione, convertita analogicamente dal DAC tuttofare, è inviata al processo.

Dopo l'eventuale polling di altri porti per altre funzioni, il ciclo ricomincia.

Chi è abbastanza familiare con la teoria dei controlli numerici può provare a fare un programma che segua le linee or ora indicate.

Noi, molto più banalmente ci muoveremo invece nell'ipotesi di attuare col microprocessore il tipo più semplice di controllo, quello detto solitamente ON-OFF o "*a tutto-niente*".

Programma P. 10-4

CONTROLLO DI TEMPERATURA A TUTTO-NIENTE

Il metodo consiste nel tenere in memoria due valori, denominiamoli fin d'ora VALINF e VALSUP, *entro i quali* si desidera che la temperatura (fissiamo questa grandezza solo per comodità di riferimento) si mantenga.

Il micro controlla, in ogni ciclo, il valore fornito dal trasduttore: se esso supera il limite superiore invia un comando per la *chiusura* di una valvola (esempio: quella di immissione carburante al bruciatore); qualora invece si dovesse essere al di sotto del limite inferiore la valvola viene aperta.

È chiaro allora che, oltre ad essere molto semplificata la parte di elaborazione della correzione, in queste ipotesi della scheda di figura 10-8 lo switch analogico 1 (e la V_{out}) non servono più.

Per implementare il nostro programmino, supponiamo di utilizzare, oltre al porto 0 cui è connesso il DAC i seguenti porti:

- *porto 2* = porto di controllo del porto 0 della PIO;
- *porto 3* = porto di controllo del porto 1;
- *porto 5* = porto per l'immissione dall'esterno di nuovi valori.

Si può infatti pensare che si voglia poter stabilire in esercizio, nuovi limiti entro i quali l'uscita debba tenersi (altrimenti il nostro impiantino avrebbe una ben misera flessibilità...). A proposito di questo porto non facciamo ipotesi, limitandoci a dire che esso è di input, ed è collegato ad opportuno dispositivo tipo tastiera o, come vedremo da ultimo, un commutatore più pulsante.

I bit del porto 0, che lavorano in control mode, hanno le funzioni seguenti:

- *bit 7* = di input, uscita del comparatore, come già visto;
- *bit 6* = di output, linea di comando del Sample & Hold;
- *bit 5* = di read out sui LED, che qui immaginiamo presenti;

- *bit 4* = segnale per l'avvio processo, di output;
- *bit 3* = input, segnalazione di "processo avviato", proveniente dall'impianto;
- *bit 2* = di output, comando apertura valvola bruciatore;
- *bit 1* = output, per la chiusura della detta valvola;
- *bit 0* = input: è alto quando si supera un limite assolutamente, pericoloso.

Per realizzare quest'ultima funzione si può pensare di usare un altro comparatore, sull'entrata non invertente del quale è connessa la V_{tr} del trasduttore, mentre sull'ingresso invertente del comparatore è posta una tensione di riferimento (realizzata per esempio con un diodo Zener ed una resistenza) calcolata in modo da corrispondere a quel valore (estremo) che si considera intollerabile. In questa evenienza, dovrà essere messo in opera un interrupt per il più rapido azionamento possibile della chiusura valvola, gli allarmi del caso ecc.

Quanto ai bit 4, 2 e 1 si tratta di azionamenti impulsivi richiedenti tutti un medesimo tempo di durata impulso (quello, mettiamo, per l'eccitazione di un teleruttore).

Sulla base di questa descrizione, il lettore, confrontando con la figura 10-7, è perfettamente in grado di ricostruire lo schema dell'hardware, onde non riproduciamo alcuna figura (anche perchè il disegnatore è andato in ferie, ormai stanco).

Il flow chart segue grossomodo le linee di quello di figura 10-8, pertanto diamo senz'altro qui di seguito la codifica Assembly, limitandoci a precisare che il comando 4 è inviato inizialmente con funzioni di accensione o simili, che richiedono un certo tempo per la fine del quale si è pensato di affidare il compito di segnalazione ad un dispositivo esterno opportuno, connesso come si è detto, al bit 3 dell'ormai celebre porto 1.

Al solito, mettiamo le mani avanti circa la schematicità dell'esempio che, anche se vuol riassumere varie cosettine è, oltre che sommario, abbastanza incompleto rispetto a casi reali, mancando ad esempio un comando per l'arresto "regolare", a fine esercizio, dell'impianto (questa funzione però si può sempre pensare affidata all'hardware esterno oppure...al solito lettore paziente-perfezionista).

Vediamo passo passo le varie istruzioni.

Parte introduttiva dalla label VALINF alla MARCIA (esclusa).

Si definiscono con delle EQU: i limiti inferiori e superiori (VALINF e VALSUP) *medi*, cioè relativi ad una condizione di funzionamento più frequente;

Programma P. 10-4

Label	Codice Assembly	Commenti
VALINF	EQU xxxx	limite inferiore (medio)
VALSUP	EQU yyyy	limite superiore (medio)
RITIMP	EQU zz	durata impulso relè
RIT 2	EQU ss	ritardi di
RIT 3	EQU tt	debouncing e/o ass.to
	ORG 0H	(avvio con comando Reset)
ZIALIZ	LD SP,400H	inizializza Stack
INIPIO	MACRO	*macro* p.1= mode 3;p.0=m.1
	JR VETTOR + 2	passa oltre vett. di interr.
VETTOR	DEFW MAMAIUT	attenti:VETTOR= indir.*pari*!
MARCIA	LD A,00010100B	set:bit: 4= avvio proc.; 2 = apri valvola
	OUT (1),A	
	LD BC,RITIMP	attendi durata imp. relè
	X OR A	reset di bit 4 e 2
	OUT (1),A	fine impulsi ai 2 relè
ATTENDI	LD BC,RIT2	
	CALL RITARD	attenti un pò proc. avviato
	IN A,(1)	
	BIT 3,A	è in marcia il bruciatore?
	JR Z,ATTENDI	se non ancora, attendi
	LD H,VALSUP	
	LD L,VALINF	
INPUT	IN A,(5)	leggi porto dei rifer.ti
	CP FFH	segnale di cambio valore?
	JR NZ,ADCONV	se NO vai a ADCONV
	LD BC,RIT3	attenti ass.to nuovo val.
	CALL RITARD	
	LD H,A	caricalo in H
SU	IN A,(5)	attendi segnale
	CP FFH	FFH
	JR NZ,SU	(abbi fede: deve arrivare)
	LD BC,RIT3	ass.to val.inf.
	CALL RITARD	poi
	LD L,A	val.inf. in reg. L
	ADCONV	macro come P. 10-3 (senza RET e risult. in D:E= com.)
	IN A,(1)	
	SET 5,A	SET
	OUT (1),A	e
	RES 5,A	RESet bit di read-out
	OUT (1),A	
CORREZ	LD A,D	valore convert.(pentito?)
	CP H	cfr.to con Val.sup.
	CALL NC,CHIUDI	se superato, chiudi valv.
	CP L	cfr. con Val. inf.

Label	Codice Assembly	Commenti
	CALL C, APRI	se troppo basso, apri v.
	JP INPUT	ricomincia il ciclo
APRI	IN A,(1)	
	SET 2,A	manda un impulso
	OUT (1),A	sul bit di apertura valv.
	LD BC,RITIMP	per la durata RITIMP
	CALL RITARD	di comando relè
	RES 2,A	
	OUT (1),A	
	RET	
CHIUDI	IN A,(1)	impulso
	SET 1,A	di
	OUT (1),A	durata
	LD BC,RITIMP	RITIMP
	CALL RITARD	al relè di
	RES 1,A	comando
	OUT (1),A	chiusura valvola
	RET	
RITARD	. . . . . .	solita routine con BC
	RET	decremento
MAMAIUT	CALL CHIUDE	chiudi valvola!
	. . . . . .	ad libitum
. . . . . .	. . . . . .	seguono MACRO
	END	

RITIMP è il ritardo, misurato in una certa unità KDEL (interna a RITARD e sottointesa), il quale corrisponde al tempo ottimale per i relè di azionamento avvio processo (bit 4, porto 1), apertura (bit 2) e chiusura (bit 1) bruciatore o altro elemento riscaldante; RIT2 e RIT3 sono invece ritardi di debouncing e assestamento.

La ZIALIZ provvede ad inizializzare lo Stack Pointer in fondo alla memoria RAM (implicita una dimensione complessiva di memoria EPROM + RAM di 1024 =400H byte, in casi diversi va ovviamente cambiato tale valore). La pseudo istruzione ORG fissa invece l'inizio del programma all'indirizzo 0000H, per avere così la possibilità di far partire il tutto con il comando di Reset inviato con qualche mezzo (manuale o automatico) alla CPU. La MACRO denominata INIPIO provvede, secondo modalità per le quali rimandiamo all'apposito capitolo 7, a fissare sui porti 0 e 1 della unità PIO il modo 1, input, sul porto 0 e il modo 3, I/O con bit 0 di

interrupt vettorizzato, sul porto 1. A questo punto, prima di definire la label del vettore di interrupt VETTOR, per il cui indirizzo, ricordarselo sempre!, *è necessario un valore pari* (far bene i conti ed aggiustare se del caso con una istruzione NOP) si è fatto un salto relativo a VETTOR + 2 (andava naturalmente anche bene JR MARCIA ma così si è pensato di drammatizzare di più). Questo serve in quanto, nella posizione in cui si è posto VETTOR il contenuto della voce contiene *un dato* e guai se venisse interpretato come una istruzione (!!!).(°)

Parte iniziale: da MARCIA a INPUT (inclusa)

Si comincia con il settare i bit 4 di avvio processo ed il 2 per l'apertura valvola o similari (gli zero si riferiscono o a comandi da non attivare oppure a bit indifferenti cioè di input); il bit 5 di read out in questa fase di avvio, per il modo in cui è stato impostato l'hardware non serve a nulla, dato che sul porto 0, finchè non c'è in funzione la conversione A/D (perfezionisti,all'opera). Al termine di RITIMP il reset dei comandi 4 e 2 avviene semplicemente con XOR A che azzera l'accumulatore, tanto qui tutti i bit sono zero, con risparmio evidente (rispetto a due istruzioni RES si risparmiano due byte) (°°)

Dopo la label ATTENDI seguono istruzioni mediante le quali si aspetta l'arrivo sul porto 1 di un valore alto del bit di segnalazione "*processo avviato*": si tratta di un procedimento sostanzialmente già visto in questo e nel precedente capitolo, perciò non insistiamo.

Le istruzioni dopo la label INPUT funzionano nell'ipotesi che l'hardware sul porto 5 sia progettato in modo da inviare, al momento in cui viene scritta una coppia di valori nuovi VALSUP e VALINF per due volte un segnale di premonizione FFH. La routine con i suoi bravi commenti dovrebbe essere self evident per chi è ormai giunto a queste ultimissime pagine. Costui potrà anche notare, rispetto ad analogo esempio precedente una certa semplificazione e, magari qualche sia pur lieve inconveniente. Se il segnale FFH non è inviato dalla periferica sul porto , si salta, senza por tempo in mezzo alla parte seguente (ADCONV) in cui, finalmente, il controllo di processo funziona.

(°) - Si poteva anche mettere VETTOR proprio in fondo, ma così sarebbe forse stato tutto meno chiaro. Si osservi che all'inizio, dato che si è supposto un comando Reset, occorre, ovunque sia posto VETTOR, un salto oltre la voce.

(°°) - Nel caso in cui si fossero dovuti mantenere dei bit di valore uno, diversi da 4 e 2, si poteva fare il reset *soltanto di questi* con l'istruzione: AND EBH, cioè l'and con l'immediato 11101011B, di ovvio significato, risparmiando ancora due byte.

Parte centrale: da PROSEG ad APRI

Si opera a questo punto la conversione software A/D con le medesime modalità della omonima routine del programma P. 10-3. Sole varianti, la mancanza della RET e i registri D ed E al posto rispettivamente di B e C (dato che qui crescono registri; ma si poteva anche lasciare inalterata la ADCONV). Al termine di questa fondamentale operazione, il valore ora esistente sul porto 0 è scritto sul gruppo dei LED a mezzo di un impulso (di microsecondi, più che sufficiente per i latch elettronici), quindi si opera la semplicissima correzione basata sul confronto tra il valore convertito, in D (°) e, una prima volta con VALSUP contenuto in H, l'altra con L che contiene VALINF. Nei due casi, se, rispettivamente si è al di sopra e al di sotto di tali limiti si operano le correzioni più che evidenti di cui alle routine CHIUDI e APRI, rispettivamente. A proposito delle quali,l'unica osservazione che ci sentiamo di fare è che non occorre resettare il comando chiusura oppure apertura valvola, dato che, con il funzionamento impulsivo entrambi i comandi stanno alti unicamente per la durata corrispondente a RITIMP.

A questo punto operata, se necessario, la correzione, con il salto a INPUT si ricomincia il ciclo per il polling del porto 5.

Circa le routine, non abbiamo da dire molto a riguardo. Per la MAMAIUT (è l'ultima boutade...) dopo l'obbligatoria CALL CHIUDI il lettore potrà immaginare ogni genere di istruzioni, dal lampeggio minaccioso dei LED allo strombazzare di tutti i cicalini e sirene che gli sembrino opportuni).

Circa la routine di interrupt, come l'abbiamo impostata, semplicemente, le cose vanno così: quando la V_{tr} supera la V_z si salta a MAMAIUT che chiude la valvola e, come è opportuno, lascia passare in segnalazioni un certo tempo. Questo può essere anche relativamente piccolo (comunque non superiore a quello del naturale cessare del fenomeno, se momentaneo). Può accadere che si continui ad uscire e rientrare nella routine e l'allarme si manifesterà esternamente con carattere continuativo. Ad ogni rientro dalla MAMAIUT si avanzerà di un'istruzione nel programma vero e proprio, senza che questo rechi inconvenienti seri (a parte l'inattendibilità, in questa fase, del read-out), dal momento che ora la comparazione con VALSUP dovrà dare per forza condizioni che invocano le operazioni di CHIUDI.

(°) - E anche, se si è compreso il meccanismo della routine ADCONV, sull'uscita del porto 5, su cui il valore di B è caricato *prima* di uscire con Cy = 1 dal sottoprogramma.

Naturalmente le cose si possono complicare in vari modi: per esempio si potrebbe prevedere un diverso rientro se le condizioni di allarme perdurano per troppo tempo (con un contatore del numero di chiamata di MAMAIUT, supponiamo); a tale proposito si può seguire la tecnica già vista nel Cap. 7 o, più semplicemente, utilizzare un deviatoio (es. Cy) che, interrogato in un punto qualsiasi del MAIN (tanto è ciclico, quindi questo test è forzatamente operato), comporti la chiamata di una subroutine di "ripristino" opportuna. E chi più ne ha.......

Variante alla label INPUT

Si supponga che i valori di temperatura che interessano in una certa applicazione siano solo 4 e, allo scopo di semplificare il discorso, che questi valori siano rappresentabili in binario dai numeri: 10000000, 01000000, 00100000 e 00010000. Questo è sempre possibile, facciamo notare: basterà opportunamente tarare il blocco trasduttore e il guadagno del DAC in modo che ai quattro valori effettivi espressi in °C corrispondano, mettiamo (ma solo per fissare le idee) i valori 12,8V, 6,4V, 3,2V e 1,6V che il DAC fornisce in uscita quando in input ha i dati binari or ora detti (che in decimale sarebbero 128; 64; 32 e 16).

L'hardware esterno può in questo caso essere costituito semplicemente da un commutatore a 4 posizioni per mandare a scelta il livello alto sul bit 7, 6, 5 o 4 del porto 5 seguito da un segnale, mettiamo, sul bit 0 del medesimo porto tramite pulsante: così, quando questo segnale arriva, il dato nuovo è già stabile sul commutatore. Se chi legge ha seguito bene il discorso capirà che con questo sistema si introduce il valore *ideale*, anzichè i due limiti inferiori e superiori. Questi allora dovranno essere calcolati dal bravo Z80. E, nelle ipotesi che siamo andati facendo non è neanche difficile. Basta osservare che i dati limiti saranno di una stessa percentuale in più o in meno di quello ideale introdotto e, nella maggior parte dei casi, sarà opportuno siano il più vicini possibile tra di loro.

Parlando in generale quindi, il microprocessore dovrebbe calcolare, con una moltiplicazione questi limiti. Ma nel nostro caso non occorre.

Infatti per il valore più basso, 00010000, i valori più vicini possibili, compatibili cioè con la risoluzione consentita dagli otto bit, saranno: 00001111 e 00010001, si aggiunge un'unità. Per la proporzionalità detta, allora, agli altri valori andranno aggiunti e tolti rispettivamente: 10B per 32; 100B per 64 e 1000B per 128.

Sulla base delle precedenti considerazioni si può impostare un algoritmo consistente nel testare, dopo la lettura del porto 5, il bit 4, facendo successivamente circolare *verso destra* l'accumulatore aggiungendo al dato originario (depositato in D dopo l'input) il contenuto del registro E, inizialmente posto a 00000001 e anch'esso oggetto di rotazione *sinistra* con la detta

somma che è messa in atto quando infine il bit 4 è trovato uguale a 1.

La modifica assume allora l'aspetto seguente (si è pensato per semplicità che sul pulsante connesso al bit 0 del porto 4 ci sia affidabile debouncer hardware del tipo classico formato da un flip flop SR):

```
INPUT       IN A,(5)
            BIT 0,A          com'è il bit 0 sul porto 5?
            JR Z,ADCONV      se nullo, vai alla MACRO
            RES 0,A          se = 1, annullalo (non c'entra col dato)
            LD D,A           dato (128:64...) in depodito D
            LD E,1           E = 00000001
TESTB4      BIT 4,A          bit 4  = 1? (cioè dato = 16, poi 32 ecc)
            JR NZ,INFSUP     se SI va a calcolare VALSUP VALINF
            RRCA             metti bit 5 in bit 4 ecc.
            RCL E            fai E = 00000010 (poi 00000100 ecc.)
            JR TESTB4        torna al test del bit 4
INFSUP      LD A,D           ricupera dato letto
            SUB E            valore inferiore
            LD L,A           nel reg. L
            LD A,D
            ADD E            valore superiore
            LD H,A           in H
            ADCONV           eccetera.....
```

E, PER FINIRE: UN PROGRAMMA ... FUORI PROGRAMMA

Programma P. 10-5

EMULAZIONE DI RETE LOGICA COMBINATORIA

Si utilizzano i porti 4 e 5, il primo come input, l'altro come output.

Del porto 4 i bit 7,6,5 e 4 non sono utilizzati (meglio, sono a zero), mentre i bit 3,2,1 e 0 costituiscono le variabili di ingresso della rete logica combinatoria che il nostro Z80 è chiamato ad emulare. A questi bit andranno connessi interruttori e/o pulsanti. Sul bit 7 del porto 5, di output, andrà invece attaccato un LED in guisa tale che esso si accenda quando tale bit 7 è "on".

Il programmino farà si che il microprocessore emuli una qualunque rete combinatoria a 4 bit che, nella parte iniziale di esso verrà impostata tramite la tastiera (solita routine KBD). Qui per emulazione intendiamo qualcosa di più di una semplice simulazione o imitazione di funzionamento. In altri termini, con questo programma, che pure rimane nell'ambito degli esempi didattici-ludici, si può pensare che il microprocessore *sostituisca* una qualsiasi rete logica. L'impostazione del programma è fatta poi in modo che un

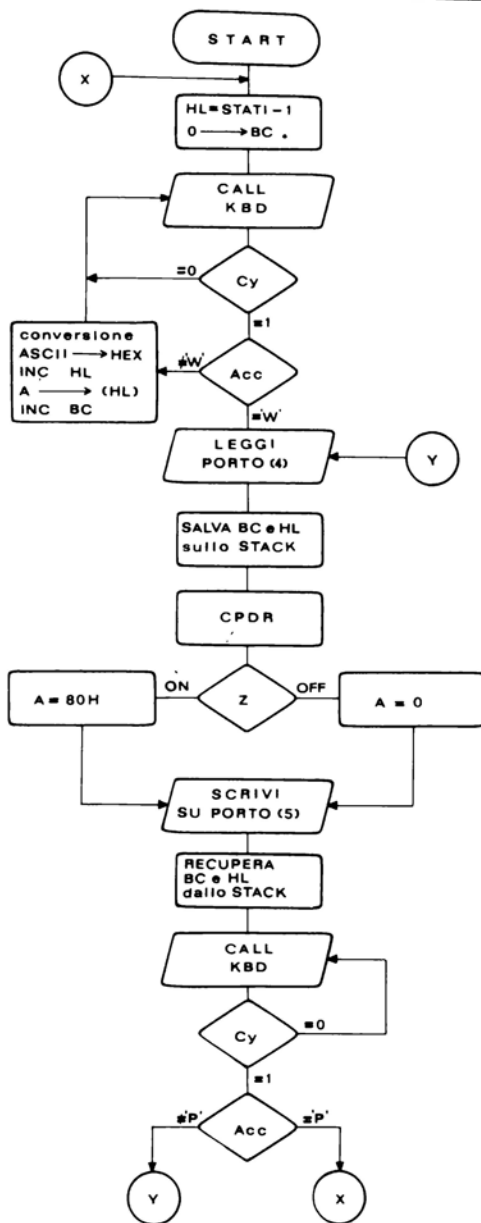



Figura 10-9 Emulazione di rete logica a 4 bit.

nuovo valore dell'uscita si abbia solo dopo la battuta del tasto "W" (come dire "work") che in tal modo funge da comando di *strobe o enable* della rete (così facendo, oltretutto, si eliminano problemi di debouncing in quanto prima di battere W si possono posizionare gli interruttori come si desidera).

Come si può vedere dal flow-chart di figura 10-9, nella prima parte del programma si caricano successivamente gli STATI della funzione booleana desiderata in cui si vuole uscita "1". Come è noto, per stato di una "*tabella di verità*" si intende ogni configurazione binaria delle variabili di input. Nel caso di quattro variabili gli stati sono sedici, da 0000 a 1111. Si userà, per immagazzinare gli stati in cui il bit 7 di uscita si chiede sia "1", una tabella in cui il numero di elementi è, per completezza, proprio sedici ma il cui sedicesimo elemento solo teoricamente potrà essere utilizzato: infatti l'avere tutti gli stati con uscita "1" equivale a definire una funzione identica che non trova evidentemente pratica applicazione. Per chiarire le modalità operative, riferiamoci ad un esempio qualsiasi:

sia da emulare una funzione booleana che vale "1" in corrispondenza degli stati 0010, 0110, 1010 e 1100. Per caricare la tabella, che inizia alla label denominata STATI, si batteranno successivamente i tasti corrispondenti al valore esadecimale di ciascuno dei detti stati e cioè 2, 6, A e C.

Inizialmente si punta con HL a STATI - 1 e si pone a zero il "Byte Counter" BC. Segue la solita KBD con attesa della battuta. Questa è comparata con l'ASCII "W" e, se l'Accumulatore non contiene tale carattere, dopo incremento di HL e BC, si carica in (HL) il dato in A. (°) Prima di caricare in (HL) però il dato in accumulatore subisce una correzione da ASCII (codice fornito dalla routine KBD) ad esadecimale: ad es. un dato come 32H deve essere trasformato in 02H, mentre un dato come 42H (lettera 'B') deve diventare invece OBH.

Vediamo ora cosa avviene nella seconda parte del programma cui si salta con il comando "W". La lettura del porto di input è seguita dal salvataggio sullo stack di BC e HL, quindi si invoca la possente istruzione CPDR (Compara Decrementa e Ripeti) finchè o BC = 0 oppure un qualche (HL) concorda con il contenuto dell'Accumulatore, cioè con lo stato degli input testè letto dal porto 4. Due osservazioni a riguardo: si è ritenuto conveniente l'uso della comparazione ripetuta "a ritroso", cioè partendo dall'ultima posizione della tabella STATI in cui nella fase di *programmazione* della funzione booleana è stato caricato un elemento; evitando così di riposizionare

---

(°) - Si noti come questo modo di procedere da un lato consenta di definire anche la funzione identità con uscita sempre nulla (quando si batte subito W), dall'altro permette di uscire con HL che punta sempre all'ultimo valore caricato e BC esattamente uguale al n° elementi (anche zero).

HL all'inizio; al termine della CPDR è il flag Z ad indicare condizione di "*trovato*" quando Z = 1, mentre Z = 0 denota la situazione "*non trovato*".

L'azione successiva da compiersi nei due casi è allora evidente: se uno stato della tabella corrispondente alla condizione degli interruttori di input letti dal porto 4, per quanto si è detto, ciò significa che l'uscita deve valere "1", viceversa, sulla condizione Z = 0, sarà da porre zero in output: ciò viene fatto ponendo, nel primo caso 80H, cioè bit 7 = 1 e tutti gli altri bit nulli, nel secondo caso zero in Accumulatore, facendo poi subire dopo l'output di A sul porto 5. Con POP HL e POP BC si ripristinano i valori che si avevano prima della CPDR.

A questo punto si interroga ancora la KBD e, con Cy = 1, si controlla se in Accumulatore c'è il carattere "P". In caso negativo, ossia è stato di nuovo premuto "W" si torna a leggere (4) e alla comparazione per moto contrario partendo dai valori di HL e BC già salvati poi ripristinati.

Qualora invece si desideri passare ad un'altra tabella di verità, con la battuta di "P" si torna, tramite il connettore X, al punto in cui si carica, a partire dall'indirizzo etichettato con la label STATI, il nuovo insieme di stati in cui la nuova funzione booleana si desidera dia uscita "1".

Volendo, si può anche eliminare, in momenti successivi alla fase di caricamento tabella degli stati, il comando di "strobe" costituito dal tasto "W"; basterà far seguire la chiamata di KBD che è disegnata in fondo al flow dalla sola interrogazione di Cy; con Cy = 1 (il che significa tasto "p" premuto, ma ora *qualsiasi tasto va bene*) si salterà senz'altro al punto X, mentre con Cy = 0 anzichè ri-chiamare la KBD ci si collegherà ad Y per una (continua e ripetuta, in "polling") lettura del porto 4 di input.

Codifica

Dopo il flow chart e le spiegazioni or ora date essa non presenta problemi di sorta. L'unica osservazione che riteniamo di fare è la tecnica seguita per la conversione da ASCII a Hex.: il dato in Accumulatore è comparato con 'A', cioè con 41H (v. TAV. 1-III cap. 1) e, se minore- il che implica cifra da 0 a 9 - si salta a COMCORR, altrimenti - cifra da A a F - si aggiunge 9H all'Accumulatore. In questo caso, se, ad esempio, si ha inizialmente Acc. = 42H, si ottiene poi: (42+9)H = 4BH, cioè abbiamo portato nel semibyte destro il valore corretto B. A questo punto, con l'istruzione (esistente alla label denominata COMCORR, cioè COMune CORRezione) AND 0FH viene cancellata (e sostituita con 0) la inutile zonatura 3 o 4 che sia.

Il lettore confronti questa tecnica con quella proposta al Capitolo 5 (programma P. 5-5) e ne concluderà che molte strade portano a Roma (tranne quelle ... sbagliate) e che tutto è perfettibile.

Programma P. 10-5

Ind.	Contenuto	Label	Codice Assembly	Commenti
			ORG D00	
0D00		STATI	DEFS 16	
D10	3E 7F		LD A,7FH	modo1
D12	D3 06		OUT (6),A	su porto 4
D14	3E 0F		LD A,0FH	modo 0
D16	D3 07		OUT (7),A	su porto 5
D18	AF		XOR A	poni zeri
D19	D3 O5		OUT (5),A	su porto 5
D1B	01 00 00	DABORD	LD BC,0	Byte Counter = 0
D1E	21 FF OD		LD HL,STATI-1	HL punta a sin. tab STATI
D2	CD 69 00	ATTKEY	CALL 69H	chiama la KBD
D24	30 FB		JR NC,ATTKEY	se Cy = 0 torna ATTesa KEY
D26	FE 57		CP 'W'	è l'ASCII 'W'?
D28	28 0D		JR Z,WORK	se si, al lavoro!
D2A	FE 41		CP 'A'	cfr con 'A' = 41H
D2C	38 02		JR C,COMCORR	se minore, va a COMCORR
D2E	C6 09		ADD A,9	sennò aggiungi 9:
			;es. (42+9)H = 4BH	
D30	E6 0F	COMCORR	AND 0FH	azzera semibyte sinistro
D32	23		INC HL	passa a STATI, STATI+1 ecc.
D33	77		LD (HL),A	carica stato in cui U = 1
D34	03		INC BC	increm. Byte Counter
D35	18 EA		JR ATTKEY	altro tasto
D37	DB 04	WORK	IN A,(4)	leggi porto 4
D39	C5		PUSH BC	salva Byte Counter
D3A	E5		PUSH HL	e puntatore HL
D3B	ED B9		CPDR	fai ricerca tab. a ritroso
D3D	3E 00		LD A,0	A = 0 (v. nota)
D3F	CC 52 0D		CALL Z,SETON	se Z =1 fai A = 80H
D42	D3 05		OUT (5),A	on o off bit 7 porto 5
D44	E1		POP HL	ricupera dello stack HL
D45	C1		POP BC	e BC
D46	CD 69 00	KEYB2	CALL 69H	(battere 'W' oppure 'P'
D49	30 FB		JR NC,KEYB2	
D4B	FE50		CP 'P'	altra tabella stati?
D4D	20 E8		JR NZ,WORK	se NO, lavora ancora
D4F	C3 1B 0D		JP DABORD	se SI, va a ricaricare la nuova tabella
D52	3E 80	SETON	LD A,80H	bit A7 in on
54	C9		RET	

Nota: non andava bene l'istruzione XOR A, in quanto essa, oltre ad azzerare l'Accomulatore, pone anche Z = 0 (mentre LD A,0 *conserva* il valore di Z, onde la successiva CALL Z,SETON funziona correttamente).

COMMIATO

Molti sono al solito gli esperimenti che chi dispone di un sistema di sviluppo come il NASCOM o simili può effettuare, aggiungendo un pò di hardware in base agli esempi qui fatti: conversione A/D voltmetro digitale per c.c., controlli vari sia collegandosi ad un vero processo(o ad una sua riproduzione in scala, tipo impianto pilota) come pure semplicemente inviando segnali che simulino la risposta del sistema controllato vero e proprio.

Anzi questo è di solito ciò che è consigliabile fare in sede di debug. Su tali argomenti comunque non possiamo che fare questi fugaci cenni.

Terminiamo il testo con un ultimissimo trucchetto a proposito della routine RITARD varie volte citata. Si è pensato di sfruttare l'istruzione CPI. Qualcuno si domanderà: ma cosa c'è da comparare? Nulla, però, dato che prima di entrare in RITARD si posiziona BC, noi "inganniamo" lo Z80 facendogli fare, con la CPI il decremento automatico di BC e (ma di questo a noi non interessa nulla, almeno in tale caso) il confronto tra quel che v'è in (HL) (e che HL punti pure dove vuole) e quello che c'è in Accumulatore. Quello che pure ci sta a cuore è invece il fatto che, con BC = 0 la CPI attiva la condizione PO (Parity Odd) e questo sì che ci va a genio. Conclusione:

```
RITARD      CALL KDEL       5 msec. (o altro)
            CPI             compara come ti pare!
            JP PE,RITARD    se BC≠0 (PE =1) RITARDa
            RET             esci con BC = 0
```

Cortina, vero? (°)

IL LIBRO FINISCE QUI MA IL DISCORSO CONTINUA

(°) - Contro i facili entusiasmi, avvertiamo però che, dato che la CPI influenza HL, qualora sia necessario salvare HL, l'aggiunta di un PUSH HL e di un POP HL allungano anche questo in altri casi allettante programmino.

DESTINATION		SOURCE: REGISTER							SOURCE: IMM. EXT.	SOURCE: EXT. ADDR.	SOURCE: REG. INDIR.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
REGISTER	AF										F1
REGISTER	BC								01 n n	ED 4B n n	C1
REGISTER	DE								11 n n	ED 5B n n	D1
REGISTER	HL								21 n n	2A n n	E1
REGISTER	SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
REGISTER	IX								DD 21 n n	DD 2A n n	DD E1
REGISTER	IY								FD 21 n n	FD 2A n n	FD E1
EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
REG. IND.	(SP)	F5	C5	D5	E5		DD E5	FD E5			

PUSH INSTRUCTIONS → (row REG. IND. (SP))

POP INSTRUCTIONS ↑ (column REG. INDIR. (SP))

Tabella A-2. Istruzioni di caricamento a 16 bit, più "PUSH" e "POP".

		IMPLIED ADDRESSING				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

Tabella A-3. Scambi "EX" e "EXX".

			SOURCE	
			REG. INDIR.	
			(HL)	
DESTINATION	REG. INDIR.	(DE)	ED A0	'LDI' – Load (DE)←(HL) Inc HL & DE, Dec BC
			ED B0	'LDIR,' – Load (DE)←(HL) Inc HL & DE, Dec BC, Repeat until BC = 0
			ED A8	'LDD' – Load (DE)←(HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' – Load (DE)←(HL) Dec HL & DE, Dec BC, Repeat until BC = 0

Tabella A-4. Trasferimento blocchi.

SEARCH LOCATION	
REG. INDIR.	
(HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR', Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Repeat until BC = 0 or find match

Tabella A-5. Ricerca blocchi.

SOURCE

	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Tabella A-6. Istruzioni aritmetico-logiche (8 bit).

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

Tabella A-7. Operazioni generali su AF.

			SOURCE					
			BC	DE	HL	SP	IX	IY
DESTINATION	'ADD'	HL	09	19	29	39		
		IX	DD 09	DD 19		DD 39	DD 29	
		IY	FD 09	FD 19		FD 39		FD 29
	ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
	SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
	INCREMENT 'INC.		03	13	23	33	DD 23	FD 23
	DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Tabella A-8. Aritmetica a 16 bit.

Source and Destination

TYPE OF ROTATE OR SHIFT	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
'RLC'	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
'RRC'	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
'RL'	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
'RR'	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
'SLA'	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
'SRA'	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
'SRL'	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
'RLD'								ED 6F		
'RRD'								ED 67		

	A
RLCA	[illegible]
RRCA	[illegible]
RLA	[illegible]
RRA	[illegible]

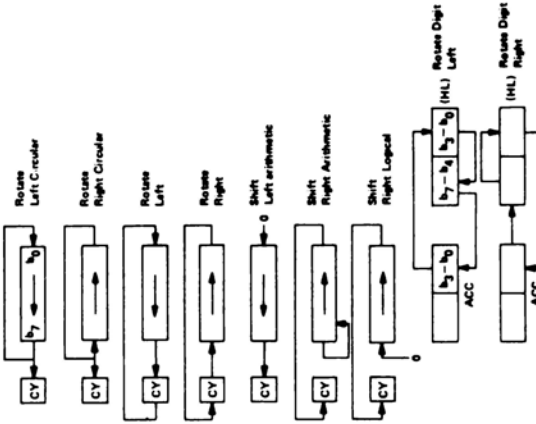



Tabella A-9. Rotazione e shift.

		REGISTER ADDRESSING							REG. INDIR.	INDEXED	
	BIT	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

Tabella A-10. Manipolazione bit.

			CONDITION									
			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B≠0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'		(HL)	E9									
JUMP 'JP'	REG. INDIR.	(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC+e										10 e-2
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

Tabella A-11. Istruzioni di salto, chiamate e rientri da subroutine.

Tabella A-13. Istruzioni di input.

				PORT ADDRESS	
				IMMED.	REG. INDIR.
				n	(C)
INPUT DESTINATION	INPUT 'IN'	REG ADDRESSING	A	[illegible] n	ED 78
			B		ED 40
			C		ED 48
			D		ED 50
			E		ED 58
			H		ED 60
			L		ED 68
	'INI' – INPUT & Inc HL, Dec B	REG, INDIR	(HL)		ED A2
	'INIR'– INP, Inc HL, Dec B, REPEAT IF B≠0				ED B2
	'IND'– INPUT & Dec HL, Dec B				ED AA
	'INDR'– INPUT, Dec HL, Dec B, REPEAT IF B≠0				ED BA

BLOCK INPUT COMMANDS ('INI', 'INIR', 'IND', 'INDR')

		OP CODE	
CALL ADDRESS	0000_H	C7	'RST 0'
	0008_H	CF	RST 8'
	0010_H	D7	'RST 16'
	0018_H	DF	'RST 24'
	0020_H	E7	'RST 32'
	0028_H	EF	'RST 40'
	0030_H	F7	'RST 48'
	0038_H	FF	'RST 56'

Tabella A-12. Istruzioni di restart.

			SOURCE							
			REGISTER							REG. IND.
			A	B	C	D	E	H	L	(HL)
'OUT'	IMMED.	n	D3 n							
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3
'OUTD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB

BLOCK OUTPUT COMMANDS

PORT DESTINATION ADDRESS

Tabella A-14. Istruzioni di output.

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM0'	ED 46	8080A MODE
SET INT MODE 1 'IM1'	ED 56	CALL TO LOCATION 0038_H
SET INT MODE 2 'IM2'	ED 5E	INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

Tabella A-15. Istruzioni varie di controllo CPU.

APPENDICE B

Le tavole che seguono - riprodotte, come quelle della precedente Appendice A per gentile concessione della Zilog - riportano le istruzioni del microprocessore Z-80. Esse sono disposte in gruppi, nello stesso ordine che nell'Appendice A. Ciascuna tavola fornisce: l'opcode Assembly (mnemonico), il codice assoluto corrispondente, la condizione dei vari flag dopo che l'operazione è stata eseguita, il numero di byte necessari per ogni istruzione, nonché il numero di cicli di memoria e di stati T (periodi del clock esterno) necessari per il "fetch" e l'esecuzione di ognuna.

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments	
LD r, s	r ← s	•	•	X	•	X	•	•	•	01 r s		1	1	4	r, s	Reg.
LD r, n	r ← n	•	•	X	•	X	•	•	•	00 r 110		2	2	7	000	B
										← n →					001	C
LD r, (HL)	r ← (HL)	•	•	X	•	X	•	•	•	01 r 110		1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19	011	E
										01 r 110					100	H
										← d →					101	L
LD r, (IY+d)	r ← (IY+d)	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19	111	A
										01 r 110						
										← d →						
LD (HL), r	(HL) ← r	•	•	X	•	X	•	•	•	01 110 r		1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19		
										01 110 r						
										← d →						
LD (IY+d), r	(IY+d) ← r	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19		
										01 110 r						
										← d →						
LD (HL), n	(HL) ← n	•	•	X	•	X	•	•	•	00 110 110	36	2	3	10		
										← n →						
LD (IX+d), n	(IX+d) ← n	•	•	X	•	X	•	•	•	11 011 101	DD	4	5	19		
										00 110 110	36					
										← d →						
										← n →						
LD (IY+d), n	(IY+d) ← n	•	•	X	•	X	•	•	•	11 111 101	FD	4	5	19		
										00 110 110	36					
										← d →						
										← n →						
LD A, (BC)	A ← (BC)	•	•	X	•	X	•	•	•	00 001 010	0A	1	2	7		
LD A, (DE)	A ← (DE)	•	•	X	•	X	•	•	•	00 011 010	1A	1	2	7		
LD A, (nn)	A ← (nn)	•	•	X	•	X	•	•	•	00 111 010	3A	3	4	13		

										← n →				
										← n →				
LD (BC), A	(BC) ← A	•	•	X	•	X	•	•	•	00 000 010	02	1	2	7
LD (DE), A	(DE) ← A	•	•	X	•	X	•	•	•	00 010 010	12	1	2	7
LD (nn), A	(nn) ← A	•	•	X	•	X	•	•	•	00 110 010	32	3	4	13
										← n →				
										← n →				
LD A, I	A ← I	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9
										01 010 111	57			
LD A, R	A ← R	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9
										01 011 111	5F			
LD I, A	I ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9
										01 000 111	47			
LD R, A	R ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9
										01 001 111	4F			

Notes: r, s means any of the registers A, B, C, D, E, H, L
IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Tabella B-1. Caricamento a 8 bit.

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD dd, nn	dd ← nn	•	•	X	•	X	•	•	•	00 dd0 001 ← n → ← n →		3	3	10	dd Pair 00 BC 01 DE 10 HL 11 SP
LD IX, nn	IX ← nn	•	•	X	•	X	•	•	•	11 011 101 00 100 001 ← n → ← n →	DD 21	4	4	14	
LD IY, nn	IY ← nn	•	•	X	•	X	•	•	•	11 111 101 00 100 001 ← n → ← n →	FD 21	4	4	14	
LD HL, (nn)	H ← (nn+1) L ← (nn)	•	•	X	•	X	•	•	•	00 101 010 ← n → ← n →	2A	3	5	16	
LD dd, (nn)	dd_H ← (nn+1) dd_L ← (nn)	•	•	X	•	X	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	ED	4	6	20	
LD IX, (nn)	IX_H ← (nn+1) IX_L ← (nn)	•	•	X	•	X	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A	4	6	20	
LD IY, (nn)	IY_H ← (nn+1) IY_L ← (nn)	•	•	X	•	X	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A	4	6	20	
LD (nn), HL	(nn+1) ← H (nn) ← L	•	•	X	•	X	•	•	•	00 100 010 ← n → ← n →	22	3	5	16	
LD (nn), dd	(nn+1) ← dd_H (nn) ← dd_L	•	•	X	•	X	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	ED	4	6	20	
LD (nn), IX	(nn+1) ← IX_H (nn) ← IX_L	•	•	X	•	X	•	•	•	11 011 101 00 100 010	DD 22	4	6	20	

										← n →						
										← n →						
LD (nn), IY	(nn+1) ← IY_H	•	•	X	•	X	•	•	•	11 111 101	FD	4	6	20		
	(nn) ← IY_L									00 100 010	22					
										← n →						
										← n →						
LD SP, HL	SP ← HL	•	•	X	•	X	•	•	•	11 111 001	F9	1	1	6		
LD SP, IX	SP ← IX	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	10		
										11 111 001	F9					
LD SP, IY	SP ← IY	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	10		
										11 111 001	F9				qq	Pair
PUSH qq	(SP-2) ← qq_L	•	•	X	•	X	•	•	•	11 qq0 101		1	3	11	00	BC
	(SP-1) ← qq_H														01	DE
PUSH IX	(SP-2) ← IX_L	•	•	X	•	X	•	•	•	11 011 101	DD	2	4	15	10	HL
	(SP-1) ← IX_H									11 100 101	E5				11	AF
PUSH IY	(SP-2) ← IY_L	•	•	X	•	X	•	•	•	11 111 101	FD	2	4	15		
	(SP-1) ← IY_H									11 100 101	E5					
POP qq	qq_H ← (SP+1)	•	•	X	•	X	•	•	•	11 qq0 001		1	3	10		
	qq_L ← (SP)															
POP IX	IX_H ← (SP+1)	•	•	X	•	X	•	•	•	11 011 101	DD	2	4	14		
	IX_L ← (SP)									11 100 001	E1					
POP IY	IY_H ← (SP+1)	•	•	X	•	X	•	•	•	11 111 101	FD	2	4	14		
	IY_L ← (SP)									11 100 001	E1					

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
e.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

Tabella B-2. Caricamento a 16 bit.

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE↔HL	•	•	X	•	X	•	•	•	11 101 011	EB	1	1	4	
EX AF, AF'	AF↔AF'	•	•	X	•	X	•	•	•	00 001 000	08	1	1	4	
EXX	(BC↔BC'	•	•	X	•	X	•	•	•	11 011 001	D9	1	1	4	Register bank and
	DE↔DE'														auxiliary register
	HL↔HL')														bank exchange
EX (SP), HL	H ↔(SP+1)	•	•	X	•	X	•	•	•	11 100 011	E3	1	5	19	
	L ↔(SP)														
EX (SP), IX	IX_H ↔(SP+1)	•	•	X	•	X	•	•	•	11 011 101	DD	2	6	23	
	IX_L ↔(SP)									11 100 011	E3				
EX (SP), IY	IY_H ↔(SP+1)	•	•	X	•	X	•	•	•	11 111 101	FD	2	6	23	
	IY_L ↔(SP)									11 100 011	E3				
LDI	(DE)←(HL)	•	•	X	0	X	① ↕	0	•	11 101 101	ED	2	4	16	Load (HL) into
	DE ← DE+1									10 100 000	A0				(DE), increment the
	HL ← HL+1														pointers and
	BC ← BC-1														decrement the byte
															counter (BC)
LDIR	(DE)←(HL)	•	•	X	0	X	0	0	•	11 101 101	ED	2	5	21	If BC ≠ 0
	DE ← DE+1									10 110 000	B0	2	4	16	If BC = 0
	HL ← HL+1														
	BC ← BC-1														
	Repeat until														
	BC = 0														
LDD	(DE)←(HL)	•	•	X	0	X	① ↕	0	•	11 101 101	ED	2	4	16	
	DE ← DE-1									10 101 000	A8				
	HL ← HL-1														
	BC ← BC-1														
LDDR	(DE)←(HL)	•	•	X	0	X	0	0	•	11 101 101	ED	2	5	21	If BC ≠ 0
	DE ← DE-1									10 111 000	B8	2	4	16	If BC = 0
	HL ← HL-1														

	BC ← BC-1 Repeat until BC = 0														
CPI	A – (HL) HL ← HL+1 BC ← BC-1	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 100 001	ED A1	2	4	16	
CPIR	A – (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 110 001	ED B1	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A – (HL) HL ← HL-1 BC ← BC-1	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 101 001	ED A9	2	4	16	
CPDR	A – (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 111 001	ED B9	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Tabella B-3. Scambi, trasferimento e ricerca blocchi.

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD A, r	A ← A + r	↕	↕	X	↕	X	V	0	↕	10 [000] r		1	1	4	r Reg.
ADD A, n	A ← A + n	↕	↕	X	↕	X	V	0	↕	11 [000] 110		2	2	7	000 B
										← n →					001 C
															010 D
ADD A, (HL)	A ← A+(HL)	↕	↕	X	↕	X	V	0	↕	10 [000] 110		1	2	7	011 E
ADD A, (IX+d)	A ← A+(IX+d)	↕	↕	X	↕	X	V	0	↕	11 011 101	DD	3	5	19	100 H
										10 [000] 110					101 L
										← d →					111 A
ADD A, (IY+d)	A ← A+(IY+d)	↕	↕	X	↕	X	V	0	↕	11 111 101	FD	3	5	19	
										10 [000] 110					
										← d →					
ADC A, s	A ← A+s+CY	↕	↕	X	↕	X	V	0	↕	[001]					s is any of r, n,
SUB s	A ← A - s	↕	↕	X	↕	X	V	1	↕	[010]					(HL), (IX+d),
SBC A, s	A ← A - s - CY	↕	↕	X	↕	X	V	1	↕	[011]					(IY+d) as shown for
AND s	A ← A ∧ s	↕	↕	X	1	X	P	0	0	[100]					ADD instruction.
OR s	A ← A ∨ s	↕	↕	X	0	X	P	0	0	[110]					The indicated bits
XOR s	A ← A ⊕ s	↕	↕	X	0	X	P	0	0	[101]					replace the [000] in
CP s	A - s	↕	↕	X	↕	X	V	1	↕	[111]					the ADD set above.
INC r	r ← r + 1	↕	↕	X	↕	X	V	0	•	00 r [100]		1	1	4	
INC (HL)	(HL) ← (HL)+1	↕	↕	X	↕	X	V	0	•	00 110 [100]		1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	↕	↕	X	↕	X	V	0	•	11 011 101	DD	3	6	23	
										00 110 [100]					
										← d →					
INC (IY+d)	(IY+d) ← (IY+d)+1	↕	↕	X	↕	X	V	0	•	11 111 101	FD	3	6	23	
										00 110 [100]					

DEC s	s ← s - 1	↕	↕	X	↕	X	V	1	•	← d → 101				s is any of r, (HL), (IX+d), (IY+d) as shown for INC. DEC same format and states as INC. Replace 100 with 101 in OP Code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow, P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

Tabella B-4. Aritmetica a 8 bit.

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
DAA	Converts acc, content into packed BCD following add or subtract with packed BCD operands	↕	↕	X	↕	X	P	•	↕	00 100 111	27	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	•	•	X	1	X	•	1	•	00 101 111	2F	1	1	4	Complement accumulator (One's complement)
NEG	$A \leftarrow \overline{A} + 1$	↕	↕	X	↕	X	V	1	↕	11 101 101 01 000 100	ED 44	2	2	8	Negate acc, (two's complement)
CCF	$CY \leftarrow \overline{CY}$	•	•	X	X	X	•	0	↕	00 111 111	3F	1	1	4	Complement carry flag
SCF	$CY \leftarrow 1$	•	•	X	0	X	•	0	1	00 110 111	37	1	1	4	Set carry flag
NOP	No operation	•	•	X	•	X	•	•	•	00 000 000	00	1	1	4	
HALT	CPU halted	•	•	X	•	X	•	•	•	01 110 110	76	1	1	4	
DI*	$IFF \leftarrow 0$	•	•	X	•	X	•	•	•	11 110 011	F3	1	1	4	
EI*	$IFF \leftarrow 1$	•	•	X	•	X	•	•	•	11 111 011	FB	1	1	4	
IM 0	Set interrupt mode 0	•	•	X	•	X	•	•	•	11 101 101 01 000 110	ED 46	2	2	8	
IM 1	Set interrupt mode 1	•	•	X	•	X	•	•	•	11 101 101 01 010 110	ED 56	2	2	8	
IM 2	Set interrupt mode 2	•	•	X	•	X	•	•	•	11 101 101 01 011 110	ED 5E	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop CY indicates the carry flip-flop.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation.

*Interrupts are not sampled at the end of EI or DI

Tabella B-5. Controllo CPU.

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments	
ADD HL, ss	HL ← HL+ss	•	•	X	X	X	•	0	↕	00 ss1 001		1	3	11	ss	Reg.
															00	BC
ADC HL, ss	HL ← HL+ss+CY	↕	↕	X	X	X	V	0	↕	11 101 101	ED	2	4	15	01	DE
										01 ss1 010					10	HL
															11	SP
SBC HL, ss	HL ← HL-ss-CY	↕	↕	X	X	X	V	1	↕	11 101 101	ED	2	4	15		
										01 ss0 010						
ADD IX, pp	IX ← IX + pp	•	•	X	X	X	•	0	↕	11 011 101	DD	2	4	15	pp	Reg.
										00 pp1 001					00	BC
															01	DE
															10	IX
															11	SP
ADD IY, rr	IY ← IY + rr	•	•	X	X	X	•	0	↕	11 111 101	FD	2	4	15	rr	Reg.
										00 rr1 001					00	BC
															01	DE
															10	IY
															11	SP
INC ss	ss ← ss + 1	•	•	X	•	X	•	•	•	00 ss0 011		1	1	6		
INC IX	IX ← IX + 1	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	10		
										00 100 011	23					
INC IY	IY ← IY + 1	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	10		
										00 100 011	23					
DEC ss	ss ← ss - 1	•	•	X	•	X	•	•	•	00 ss1 011		1	1	6		
DEC IX	IX ← IX - 1	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	10		
										00 101 011	2B					
DEC IY	IY ← IY - 1	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	10		
										00 101 011	2B					

Notes: ss is any of the register pairs BC, DE, HL, SP pp is any of the register pairs BC, DE, IX, SP rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown. ↕ = flag is affected according to the result of the operation.

Tabella B-6. Aritmetica a 16 bit.

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
RLCA	CY ← [7 ← 0] ← (bit 7 to bit 0) A	•	•	X	0	X	•	0	↕	00 000 111	07	1	1	4	Rotate left circular accumulator
RLA	[CY] ← [7 ← 0] ← (CY to bit 0) A	•	•	X	0	X	•	0	↕	00 010 111	17	1	1	4	Rotate left accumulator
RRCA	(bit 0 to bit 7) → [7 → 0] → CY A	•	•	X	0	X	•	0	↕	00 001 111	0F	1	1	4	Rotate right circular accumulator
RRA	(CY to bit 7) → [7 → 0] → [CY] A	•	•	X	0	X	•	0	↕	00 011 111	1F	1	1	4	Rotate right accumulator
RLC r		↕	↕	X	0	X	P	0	↕	11 001 011 00 [000] r	CB	2	2	8	Rotate left circular register r
RLC (HL)		↕	↕	X	0	X	P	0	↕	11 001 011 00 [000] 110	CB	2	4	15	r — Reg. 000 — B 001 — C
RLC (IX+d)	CY ← [7 ← 0] ← (bit 7 to bit 0) r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	11 011 101 11 001 011 ← d → 00 [000] 110	DD CB	4	6	23	010 — D 011 — E 100 — H 101 — L 111 — A
RLC (IY+d)		↕	↕	X	0	X	P	0	↕	11 111 101 11 001 011 ← d → 00 [000] 110	FD CB	4	6	23	

RL s	CY ← 7 ← 0 (rotate through carry) s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	010					Instruction format and states are as shown for RLC's. To form new Op-Code replace 000 of RLC's with shown code
RRC s	7 → 0 → CY s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	001					
RR s	7 → 0 → CY (rotate through carry) s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	011					
SLA s	CY ← 7 ← 0 ← 0 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	100					
SRA s	7 → 0 → CY s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	101					
SRL s	0 → 7 → 0 → CY s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	111					
RLD	A 7-4 3-0 7-4 3-0 (HL)	↕	↕	X	0	X	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18	Rotate digit left and right between the accumulator and location (HL).
RRD	A 7-4 3-0 7-4 3-0 (HL)	↕	↕	X	0	X	P	0	•	11 101 101 01 100 111	ED 67	2	5	18	The content of the upper half of the accumulator is unaffected

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Tabella B-7. Rotazioni e shift.

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments	
BIT b, r	$Z \leftarrow \overline{r}_b$	X	↕	X	1	X	X	0	•	11 001 011	CB	2	2	8	r	Reg.
										01 b r					000	B
BIT b, (HL)	$Z \leftarrow \overline{(HL)}_b$	X	↕	X	1	X	X	0	•	11 001 011	CB	2	3	12	001	C
										01 b 110					010	D
BIT b, (IX+d)$_b$	$Z \leftarrow \overline{(IX+d)}_b$	X	↕	X	1	X	X	0	•	11 011 101	DD	4	5	20	011	E
										11 001 011	CB				100	H
										← d →					101	L
										01 b 110					111	A
															b	Bit Tested
BIT b, (IY+d)$_b$	$Z \leftarrow \overline{(IY+d)}_b$	X	↕	X	1	X	X	0	•	11 111 101	FD	4	5	20	000	0
										11 001 011	CB				001	1
										← d →					010	2
										01 b 110					011	3
															100	4
															101	5
															110	6
															111	7
SET b, r	$r_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011	CB	2	2	8		
										$\boxed{11}$ b r						
SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011	CB	2	4	15		
										$\boxed{11}$ b 110						
SET b, (IX+d)	$(IX+d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 011 101	DD	4	6	23		
										11 001 011	CB					
										← d →						
										$\boxed{11}$ b 110						
SET b, (IY+d)	$(IY+d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 111 101	FD	4	6	23		
										11 001 011	CB					
										← d →						

RES b, s	$s_b \leftarrow 0$ $s \equiv r$, (HL), (IX+d), (IY+d)	•	•	X	•	X	•	•	•	[11] b 110 [10]					To form new O_p-Code replace [11] of SET b, s with [10] Flags and time states for SET instruction

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Tabella B-8. Manipolazione bit.

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP nn	PC ← nn	•	•	X	•	X	•	•	•	11 000 011	C3	3	3	10	
										← n →					
										← n →					cc | Condition
JP cc, nn	If condition cc	•	•	X	•	X	•	•	•	11 cc 010		3	3	10	000 | NZ non zero
	is true PC ← nn,									← n →					001 | Z zero
	otherwise									← n →					010 | NC non carry
	continue														011 | C carry
															100 | PO parity odd
															101 | PE parity even
															110 | P sign positive
JR e	PC ← PC + e	•	•	X	•	X	•	•	•	00 011 000	18	2	3	12	111 | M sign negative
										← e-2 →					
JR C, e	If C = 0,	•	•	X	•	X	•	•	•	00 111 000	38	2	2	7	If condition not met
	continue									← e-2 →					
	If C = 1,											2	3	12	If condition is met
	PC ← PC+e														
JR NC, e	If C = 1,	•	•	X	•	X	•	•	•	00 110 000	30	2	2	7	If condition not met
	continue									← e-2 →					
	If C = 0,											2	3	12	If condition is met
	PC ← PC+e														
JR Z, e	If Z = 0	•	•	X	•	X	•	•	•	00 101 000	28	2	2	7	If condition not met
	continue									← e-2 →					
	If Z = 1,											2	3	12	If condition is met
	PC ← PC+e														
JR NZ, e	If Z = 1,	•	•	X	•	X	•	•	•	00 100 000	20	2	2	7	If condition not met
	continue									← e-2 →					
	If Z = 0,											2	3	12	If condition is met
	PC ← PC+e														
JP (HL)	PC ← HL	•	•	X	•	X	•	•	•	11 101 001	E9	1	1	4	

JP (IX)	PC ← IX	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	8	
										11 101 001	E9				
JP (IY)	PC ← IY	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	8	
										11 101 001	E9				
DJNZ, e	B ← B-1	•	•	X	•	X	•	•	•	00 010 000	10	2	2	8	If B = 0
	If B = 0,									← e-2 →					
	continue														
	If $B \neq 0$,											2	3	13	If $B \neq 0$
	PC ← PC+e														

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <-126, 129>

e-2 in the op-code provides an effective address of pc+e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation.

Tabella B-9. Salti.

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	(SP-1) ← PC_H	•	•	X	•	X	•	•	•	11 001 101	CD	3	5	17	
	(SP-2) ← PC_L									← n →					
	PC ← nn									← n →					
CALL cc, nn	If condition	•	•	X	•	X	•	•	•	11 cc 100		3	3	10	If cc is false
	cc is false									← n →					
	continue,									← n →		3	5	17	If cc is true
	otherwise														
	same as														
	CALL nn														
RET	PC_L ← (SP)	•	•	X	•	X	•	•	•	11 001 001	C9	1	3	10	
	PC_H ← (SP+1)														
RET cc	If condition	•	•	X	•	X	•	•	•	11 cc 000		1	1	5	If cc is false
	cc is false														
	continue,											1	3	11	If cc is true
	otherwise														
	same as														
	RET														
RETI	Return from	•	•	X	•	X	•	•	•	11 101 101	ED	2	4	14	
	interrupt									01 001 101	4D				
RETN[1]	Return from	•	•	X	•	X	•	•	•	11 101 101	ED	2	4	14	
	non maskable									01 000 101	45				
	interrupt														

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

RST p	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$ $PC_H \leftarrow 0$ $PC_L \leftarrow p$	•	•	X	•	X	•	•	•	11 t 111		1	3	11

t	p
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

[1] RETN loads $IFF_2 \rightarrow IFF_1$

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Tabella B-10. Chiamate e rientri.

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
IN A, (n)	A ← (n)	•	•	X	•	X	•	•	•	11 011 011	DB	2	3	11	n to $A_0 \sim A_7$
										← n →					Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C)	↕	↕	X	↕	X	P	0	•	11 101 101	ED	2	3	12	C to $A_0 \sim A_7$
	if r = 110 only									01 r 000					B to $A_8 \sim A_{15}$
	the flags will														
	be affected														
			①												
INI	(HL) ← (C)	X	↕	X	X	X	X	1	•	11 101 101	ED	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1									10 100 010	A2				B to $A_8 \sim A_{15}$
	HL ← HL + 1														
INIR	(HL) ← (C)	X	1	X	X	X	X	1	•	11 101 101	ED	2	5	21	C to $A_0 \sim A_7$
	B ← B - 1									10 110 010	B2		(If B ≠ 0)		B to $A_8 \sim A_{15}$
	HL ← HL + 1											2	4	16	
	Repeat until												(If B = 0)		
	B = 0														
			①												
IND	(HL) ← (C)	X	↕	X	X	X	X	1	•	11 101 101	ED	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1									10 101 010	AA				B to $A_8 \sim A_{15}$
	HL ← HL - 1														
INDR	(HL) ← (C)	X	1	X	X	X	X	1	•	11 101 101	ED	2	5	21	C to $A_0 \sim A_7$
	B ← B - 1									10 111 010	BA		(If B ≠ 0)		B to $A_8 \sim A_{15}$
	HL ← HL - 1											2	4	16	
	Repeat until												(If B = 0)		
	B = 0														
OUT (n), A	(n) ← A	•	•	X	•	X	•	•	•	11 010 011	D3	2	3	11	n to $A_0 \sim A_7$
															Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	X	•	X	•	•	•	11 101 101	ED	2	3	12	C to $A_0 \sim A_7$
										01 r 001					B to $A_8 \sim A_{15}$

OUTI	(C) ← (HL) B ← B - 1 HL ← HL + 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 100 011	ED A3	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTIR	(C) ← (HL) B ← B - 1 HL ← HL + 1 •Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 110 011	ED B3	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 101 011	ED AB	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTDR	(C) ← (HL) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 111 011	ED BB	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Tabella B-11. Istruzioni di input/output.

L. 15.000

Cod. 327-A

L'AUTORE

Quarantenne, docente di elettronica e consulente e.d.p., ha già scritto un testo ("Fondamenti di elettronica industriale" ed. Levrotto & Bella, Torino, ora alla seconda edizione).
La sua vera attività è però la partecipazione alle remate non competitive (già due Voghelonghe di Venezia sulle spalle), nonchè la cerca dei funghi.
Ha scritto questo libercolo, su consiglio dello psicoanalista, allo scopo di guarire da una moderna nevrosi denominata "micro-mania".
La speranza di un transfert di tale complesso sull'oggetto stampato è andata completamente disillusa. Anzi, la sindrome ha subito un netto peggioramento.

GRUPPO
EDITORIALE
JACKSON

Gianni Giaccaglini

USARE IL MICROPROCESSORE

43